JBeaver
Análisis y Generación de Analizadores de Dependencias

Sistemas Informáticos 2006/07

Pedro J. Moriano Mohedano
Luis Romero Tejera
Alfonso Muñoz Moreno

director:
Pablo Gervás Gómez-Navarro

en colaboración con
Jesús Herrera de la Cruz

Facultad De Informática (UCM)

Resumen

JBeaver es un sistema de análisis y generación de analizadores de dependencias. Podemos crear corpora de entrenamiento, entrenar a un sistema automático de aprendizaje y por último realizar análisis y evaluarlos tanto gráfica como estadísticamente. Todo ello siendo éste un sistema autónomo, fácil de usar, portable, con un alto rendimiento y por supuesto, debido a su carácter didáctico y de investigación, público y gratuito. El sistema consta de tres módulos funcionales:

Entrenamiento: Crea corpora de dependencias a partir de árboles de constituyentes mediante la transformación del algoritmo de Gelbukh (Universidad Autónoma de México). Realiza el entrenamiento de la herramienta de aprendizaje automático Maltparser_0.4 por medio del corpus de dependencias, generando un modelo.Evalúa los resultados para garantizar la calidad del entrenamiento, usando las métricas: Label attachment score (LAS), Unlabel attachment score (UAS) y Label accuracy.

Análisis: Etiqueta cada token de un texto con sus part-of-speach (categorías gramaticales). Analiza el texto de entrada, ya etiquetado, gracias al modelo creado en el entrenamiento. Evalúa los resultados para garantizar la calidad del análisis, usando las métricas: Label attachment score (LAS), Unlabel attachment score (UAS) y Label accuracy.

Gráfico: Muestra en forma de árbol todas las frases que seleccionemos tanto del análisis como del entrenamiento. Existen dos formas de visualización: Por pasos (para observar su creación) Directamente.

Índice general

Capítulo 1

Introducción

En los últimos dos años ha crecido enormemente el interés que la comunidad científica dedicada al Procesamiento del Lenguaje Natural ha mostrado por los sistemas que realizan análisis de dependencias de textos.

Entre los trabajos clásicos en sistemas de análisis de dependencias es indiscutible señalar a Minipar como un referente fundamental para el inglés. Minipar ha sido una herramienta profusamente utilizada en el ámbito del Procesamiento del Lenguaje Natural, como lo demuestran las 170 citas del artículo de Dekang Lin[17] que registra actualmente el sistema Google Scholar[1].

El amplio interés que el análisis de dependencias ha suscitado en la comunidad científica, claramente influido por las notables características de Minipar, ha llevado a querer construir analizadores de dependencias para cada vez más lenguas. Aprovechando las facilidades que para esta labor proporcionan los sistemas de aprendizaje automático, recientemente se han desarrollado herramientas como MaltParser. Este tipo de sistemas son, realmente, generadores de analizadores de dependencias. Con los generadores de analizadores que basan su funcionamiento en el aprendizaje, se pueden obtener analizadores para cualquier lengua para la que se disponga de un corpus etiquetado con análisis de dependencias. En concreto, MaltParser es un sistema de análisis guiado por los datos con pretensión de ser independiente del idioma. Su creación está motivada por el hecho de que los analizadores de dependencias que están demasiado ajustados a un cierto idioma funcionan mal para otros; además, se busca también en él una solución para el indeterminismo de los analizadores de dependencias[22]. Para la fase de aprendizaje MaltParser incorpora dos posibilidades a elegir por el usuario, ambas dentro del ámbito del aprendizaje supervisado: bien mediante máquina de vector de soporte o bien mediante un modelo basado en memoria.

La existencia de Minipar y la de MaltParser suponían un estímulo importante para querer desarrollar un analizador de dependencias para el español. Gracias a MaltParser era viable, sin muchos recursos, la realización de un programa de características similares a las de Minipar, que tan buenos resultados estaba dando, que realizase análisis de dependencias para el español.

[1] `http://scholar.google.com`

Pero, además, había otro hecho que motivaba tal desarrollo: en el momento de iniciar el proyecto JBeaver ya existían analizadores de dependencias para lenguas no muy extendidas, como el sueco [22] o el turco [9]. Sin embargo, por entonces no existía un analizador de dependencias para el español, la tercera lengua más utilizada en el mundo. Por el contrario para las dos lenguas de uso más frecuente, el chino [14] y el inglés [17], ya se habían desarrollado analizadores de este tipo. Además, es notable el interés por equiparar los recursos disponibles para todas las lenguas europeas, como se puede observar en campañas como el Cross Language Evaluation Forum. Era, pues, el momento de abordar esta tarea.

Otra muestra del interés suscitado en este campo se encuentra en foros internacionales en los que se evalúan analizadores de dependencias para múltiples lenguas, como puede ser la CoNLL-X Shared Task: Multilingual Dependency Parsing. En este tipo de eventos se pueden probar las posibilidades que tienen los generadores de analizadores, como MaltParser, para producir analizadores válidos para diversas lenguas; pero no sólo eso, sino que se puede también evaluar las posibilidades de otros enfoques alternativos para abordar el problema. De este modo se ha estimulado la obtención de analizadores de dependencias para una amplia gama de lenguas; por ejemplo, en la tarea citada de la CoNLL-X se han estudiado las posibilidades actuales de realizar análisis de dependencias para los siguientes idiomas: danés, holandés, portugués, sueco, árabe, checo, búlgaro, español, alemán, japonés, esloveno, chino y turco. MaltParser obtuvo en este foro unos resultados notables, y en concreto en el análisis del español [24], para el se obtuvieron valores muy satisfactorios, del orden de los mejores obtenidos por el propio sistema y, en cualquier caso, superiores a la media de los presentados a la tarea[2].

Posteriormente a la decisión de desarrollar JBeaver, algunos grupos de investigación dieron a conocer trabajos sobre analizadores de dependencias para el español, como los de Calvo y Gelbuckh [4], Canisius et al. [5], Carreras et al.[6] Corson y Aue [7] o Nivre et al. [22]; la mayor parte de ellos en el ámbito de la CoNLL-X Shared Task: Multilingual Dependency Parsing. Sin embargo, la existencia de estos sistemas no impidió continuar con el proyecto JBeaver, a pesar de que su enfoque era muy similar al propuesto por Nivre et al. [22]; de hecho, las buenas características demostradas por el modelo de Nivre et al. probaban que el camino elegido para el desarrollo de JBeaver era prometedor. Esta última consideración, unida a que todos los sistemas participantes en la CoNLL-X Shared Task: Multilingual Dependency Parsing eran prototipos de laboratorio no disponibles públicamente, permitieron centrar esfuerzos en conseguir una herramienta autónoma, disponible pública y gratuitamente, de fácil instalación y uso, y que permitiese la incorporación de un analizador de dependencias para el español a todos los proyectos interesados en estas técnicas. Un factor importante para el desarrollo de analizadores de dependencias es la existencia de corpora anotados con los que apoyar esta labor. En concreto, para el español es reseñable la existencia del corpus Cast3LB, que ha sido utilizado tanto en el presente proyecto como en la CoNLL-X Shared Task: Multilingual Dependency Parsing.

[2] `http://w3.msi.vxu.se/users/jha/conllx/`

¿Para qué se utiliza el análisis de dependencias?

Tanto la información que proporciona el árbol de análisis de dependencias como la manera en que ésta queda estructurada resultan de gran interés para su utilización como subsistema de sistemas de Procesamiento del Lenguaje Natural. Así pues, han existido diversas propuestas de sistemas que utilizan el análisis de dependencias.

En la actualidad, en el tratamiento automático del lenguaje natural se utilizan distintos tipos de análisis. En el primer desafío PASCAL para el reconocimiento de Inferencia Textual, se enfoca el problema desde distintos tipos de análisis. Uno de ellos, el propuesto por Herrera et al.[10] en *Textual Entailment Recognision Based on Dependency Analysis and WordNet*, usa el análisis de dependencias como módulo para poder inferir nodos de hipotésis desde el texto. El sistema desarrollado por Herrera et al. [10] contiene un módulo que utiliza Minipar de D. Lin, un analizador de dependencias ya citado que normaliza los datos de los corpus de los textos, realiza un análisis de éstos y crea en memoria unas estructuras apropiadas para representarlo. En otros trabajos del primer desafío PASCAL se busca lo mismo que han hecho Herrera et al. [10] pero con otros métodos. Por ejemplo:

- *Textual Entailment Resolution via Atomic Propositions*, de Elena Akhmatova [2].
- *Applying COGEX to Recognize Textual Entailment.* de A. Fowler et al[1].
- *Recognizing Textual Entailment Using Lexical Similarity.* de V. Jijkoun et al [13].
- *Recognizing Textual Entailment with Tree Edit Distance Algorithms.* M. Kouylekov et al [16].
- *Textual Entailment as Syntactic Graph Distance: a rule based and a SVM based approach.* M. T. Pazienza et al [18].
- *Application of the Bleu algorithm for recognising textual entailments.* D. Pérez et al [?].
- *Robust Textual Inference using Diverse Knowledge Sources.* R. Raina et al [?].
- *An Inference Model for Semantic Entailment in Natural Language.* R. de Salvo Braz et al [?].
- *What Syntax can Contribute in Entailment Task.* L. Vanderwende et al [?].
- *Textual Entailment Recognition Based on Inversion Transduction Grammars.* D. Wu [?].

Con esta enumeración se muestran las distintas maneras con las que se ha enfocado este trabajo en concreto, el del reconocimiento de la inferencia textual. Entre ellos cabe mencionar el trabajo de Vanderwende et al. [8] que usa el análisis sintáctico con el mismo propósito que Herrera et al. [10] Se comprueba de esta manera que el análisis sintáctico de dependencias, puede suplir con eficacia al análisis de constituyentes.

En el congreso CLEF (Cross Language Evaluation Forum), un grupo de investigación de Pérez-Coutiño et al. [25]de Puebla, México, utilizan los árboles de dependencias. El objetivo de sus experimentos consistía en observar hasta que punto era rentable la influencia de una serie de características sintácticas a la hora de tomar una decisión en su sistema de Pregunta-Respuesta. Para conseguir ese objetivo, la respuesta se basaba en parte a una medida de densidad. Ésta medida era calculada viendo el numero de términos de preguntas que tienen una dependencia sintáctica con una candidata de respuesta. De esta manera, con esta medida se tomaban las decisiones para saber qué responder ante una pregunta. La aproximación demostró una mejora pequeña pero notable con respecto a ese mismo sistema sin este nuevo módulo.

En la décima conferencia de Aprendizaje Automático de Lenguaje Natural (10th CoNLL:Tenth Conference on Computational Natural Language Learning - New York City, June 8-9, 2006) la tarea compartida eran los analizadores de dependencias multilingües. Lo que se buscaba era el desarrollo de un analizador de dependencias aplicable para múltiples idiomas. Entre los propuestos estaba el que nosotros hemos usado para JBeaver, MaltParser de Nivre et al.[23]. Las relaciones de dependencia que modificaban la cabeza de la oración han sido empleadas como una útil representación en algunas tareas de modelado de lenguajes. Hasta hace pocos años los analizadores han sido aplicados únicamente a uno o dos idiomas: por lo general inglés, y la lengua materna del autor. Aún así, recientemente muchos anliazadores han sido aplicados a varios idiomas, p.e. el modelo de análisis de Collins ha sido testeado para el inglés y el checo, alemán, castellano y francés, mientras que el analizador de Nivre ha sido usado para el inglés, sueco y checo. Otro de los objetivos de la CoNLL 2006 ha sido la evaluación de usar los analizadores propuestos para varios idiomas, y comprobar su rendimiento en función de los tamaños de los bancos de datos y las complejidades gramaticales de distintos idiomas.

1.1. El análisis sintáctico de dependencias

En los estudios de Ingeniería Informática, cuando se trata el análisis sintáctico dentro del Procesamiento de Lenguaje Natural (Inteligencia Artificial) se estudia casi exclusivamente el análisis de constituyentes, también conocido como de estructura de frase. El análisis de constituyentes es el que típicamente se realiza en las clases de lingüística en la enseñanza secundaria y está caracterizado por el uso de la relación de inclusión (unos sintagmas incluyen a otros y, en el caso básico, se tienen sintagmas compuestos por unidades léxicas). Dentro de la categoría de análisis sintáctico, y como alternativa al análisis de constituyentes, disponemos del análisis de dependencias; éste se caracteriza por el uso de relaciones binarias (de dependencia) entre unidades léxicas.

Los árboles de análisis que se obtienen al realizar un análisis de dependencias y un análisis de constituyentes de un texto son sustancialmente diferentes. En el árbol de análisis de dependencias los nodos representan las unidades léxicas del texto y las aristas representan las relaciones binarias (dirigidas, intransitivas e irreflexivas) entre dichas unidades, por lo que cada nodo tiene un único padre. Sin embargo, el árbol de análisis de constituyentes sólo representa las unidades léxicas en las hojas, representando los demás

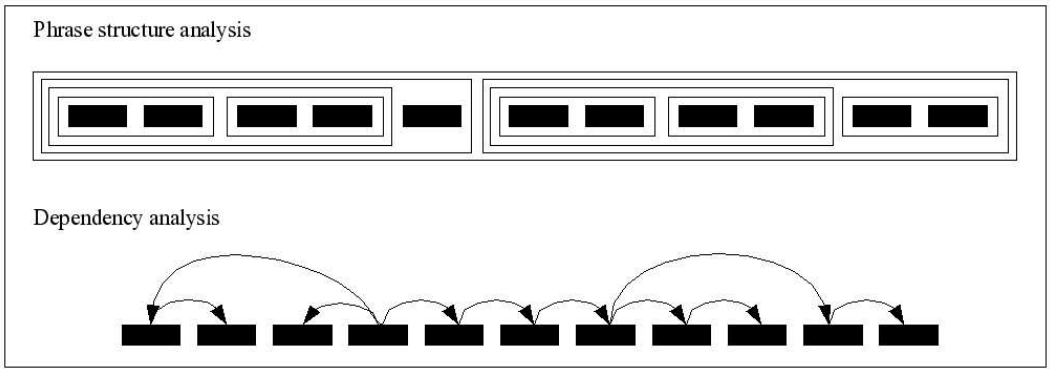

Figura 1.1: Ejemplos abstractos de un análisis de dependencias y un análisis de constituyentes (Mel'cuk, 1988)

nodos las sucesivas relaciones de inclusión entre sintagmas. En la figura 1.1 se pueden ver los árboles de un análisis abstracto de dependencias y otro de constituyentes.

¿Qué son las dependencias?

Pasemos ahora a la explicación más teórica del análisis de dependencias. Todo lo aquí explicado está sacado del artículo de I. Mel'cuk [19] y del libro de L. Tesnière [27].

Tesnière en su obra, trata acerca de los elementos de la Sintaxis Estructural. Para ello hace un estudio de la oración a nivel sintáctico. La oración es un conjunto organizado, cuyos elementos constituyentes son las palabras. Pero estas palabras han de tener cierto orden, toda palabra que forma parte de una oración deja de estar aislada, como lo está en un diccionario. Lo que une, lo que le da esa consistencia a la oración son las conexiones. Éstas son precisamente lo que nosotros tratamos. Los tipos de conexiones existentes entre las distintas palabras de una oración las distingue Mel'cuk en la clasificación siguiente:

- Dependencia semántica.
- Dependencia morfológica.
- Dependencia sintáctica

A nosotros la que nos interesa es la dependencia sintáctica. La dependencia une un elemento superior(regente) con un elemento inferior(subordinado). Cuando se quiere estudiar una frase, lo que se hace realmente es un estudio de su estructura, que es precisamente un estudio de la jerarquía de las conexiones. La forma que se tiene de representar un análisis de dependencias es mediante el llamado estema, cuya raíz, o máximo regente, estará en la parte superior del grafo, de la cual, en la mayoría de los casos, descenderán distintos subordinados, o palabras dependientes.

El análisis de dependencias se utiliza fundamentalmente en dos campos, en la enseñanza y en la investigación. En la enseñanza se utiliza para poder explicar con más

claridad las estructuras de las oraciones a analizar por el alumno. También se utiliza como paso previo a la explicación del análisis sintáctico de constituyentes de la oración. Mediante el análisis de dependencias es más sencillo ver la estructura de la oración y las dependencias entre las palabras y de esta manera entre los constituyentes.

Para el campo de la investigación, se utiliza básicamente como herramienta para sistemas de aprendizaje de lenguaje natural. Para estos sistemas se pueden utilizar distintos de analizadores, de los cuales el más asentado es el análisis de constituyentes. Desde hace algunos años, el análisis sintáctico de dependencias está cobrando fuerza, como ya se ha mostrado en el apartado anterior haciendo mención a recientes congresos y foros a nivel internacional donde el tema principal era bien el desarrollo de estos analizadores, bien su uso para funciones más complejas.

1.2. Qué es JBeaver

JBeaver es un sistema de análisis y generación de analizadores de dependencias. Podemos crear corpora de entrenamiento, entrenar a un sistema automático de aprendizaje y por último realizar análisis y evaluarlos tanto gráfica como estadísticamente. Todo ello siendo éste un sistema autónomo, fácil de usar, portable, con un alto rendimiento y por supuesto, debido a su carácter didáctico y de investigación, público y gratuito. El sistema consta de tres módulos funcionales:

- MÓDULO DE ENTRENAMIENTO

Este módulo se encarga de todas aquellas actividades realcionadas con el entrenamiento de la herramienta de aprendizaje Maltparser_0.4. Mencionar que es un módulo destinado a un uso más científico del procesamiento de dependencias y que para aquellas personas que quieran simplemente realizar análisis ya se dispone por defecto de una versión entrenada.

El módulo de entrenamiento crea corpora de dependencias a partir de árboles de constituyentes mediante la transformación del algoritmo de Gelbukh (Universidad Autónoma de México). Gracias a ese entrenamiento podemos hacer uso de la funcionalidad principal del módulo que no es ni más ni menos que la realización del entrenamiento de la herramienta de aprendizaje automático Maltparser_0.4 por medio del corpus de dependencias, generando un modelo. Una vez tenemos esto podemos evalar los resultados para garantizar la calidad del entrenamiento, usando las métricas: Label attachment score (LAS), Unlabel attachment score (UAS) y Label accuracy.

- MÓDULO DE ANÁLISIS

Este módulo tres funconalidades esenciales que deben usarse en el orden descrito a continuación: Etiqueta cada token de un texto con sus part-of-speach (categorías gramaticales). Analiza el texto de entrada, ya etiquetado, gracias al modelo creado en el entrenamiento. Y por último evalúa los resultados para garantizar la calidad del análisis, usando las métricas Label attachment score (LAS), Unlabel attachment score (UAS) y Label accuracy.

- MÓDULO GRÁFICO

Puesto que el análisis de dependencias necesita de una preparación ligüística y en muchos momentos echamos de menos poder recurrir a especialistas, se decidió la creación de un módulo gráfico que facilitara su compresión y visulización. Esto nos permite ir a cualquier experto y que nos de su opinión sobre la estructura de los árboles. Este módulo muestra en forma de árbol todas las frases que seleccionemos tanto del análisis como del entrenamiento. Y existen dos formas de visualización: Por pasos (para observar su creación) Directamente.

1.2.1. Motivación

Los objetivos del proyecto han sido los siguientes:
Seleccionar los que se han podido cumplir de la especificación inicial.

1. Evaluar la viavilidad del uso de MaltParser para obtener un analizador de dependencias para el español. En caso de que no pareciese factible, elegir una vía alternativa para obtener el analizador indicado.

2. Sintetizar con las herramientas elegidas un analizador de dependencias para el español. Así mismo, se habrá de evaluar su rendimiento. En el altamente probable caso de utilizar una herramienta que funcione por aprendizaje, desarrollar los siguientes puntos:

   a) Implementar una herramienta que permita la generación semiautomática de un corpus de análisis de dependencias para entrenar a la herramienta de aprendizaje.

   b) Entrenar la herramienta de aprendizaje para obtener un analizador de dependencias para el español.

   c) Implementar una herramienta que permita la generación semiautomática de un corpus de test para evaluar el analizador generado.

   d) Evaluar el analizador obtenido y refinarlo en función de los resultados de la evaluación.

   e) Redactar un manual de uso y referencia de la herramienta de aprendizaje (MaltParser).

   f) Establecer medidas de precisión y cobertura del analizador obtenido.

3. Evaluar la posibilidad de adaptación del analizador de dependencias a sistemas actualmente implementados.

4. Hacer disponibles en Internet los productos del trabajo realizado, así como cualquier otra actividad de difusión que se considere oportuna.

1.3. Estructura de la memoria

- El primer capítulo es esta breve introducción. En ella hemos explicado la motivación del proyecto, además de un brevísimo repaso del estado actual del análisis automático de dependencias.
- En el segundo capítulo se explica a grandes rasgos los analizadores automáticos de dependencias. Entre ellos MiniPar, que es el más extendido y usado en la actualidad, pese a su complicación y mala documentación.
- En el tercer capítulo explicamos cada uno de los recursos externos a las librerías estándar de Java que hemos utilizado para el desarrollo de la aplicación.
- El proceso de desarrollo desde el punto de vista de la Ingeniería del Software se trata en el cuarto capítulo.
- En el quinto hablamos de la evaluación del sistema.
- En el capitulo seis sacamos conclusiones del trabajo realizado, y comenzamos a perfilar el trabajo futuro, que vendría bien a medio-largo plazo.
- Por ultimo en los capítulos siete, ocho y nueve explicamos como la difusión del trabajo, citamos los agradecimientos y las referencias.

Capítulo 2

Estado actual del análisis de dependencias

2.1. Minipar

Minipar es un analizador de dependencias automático desarrollado por Dekang Lin [17] y caracterizado por obtener unas altas cobertura y precisión.

Evaluado frente al corpus SUSANNE (que es un subconjunto del Corpus Brown del Inglés Americano) elaborado por Sampson (1995) [26] , dio unos resultados del 79 % de cobertura y el 89 % de precisión; la cobertura es el porcentaje de relaciones de dependencia encontradas por el analizador del conjunto de relaciones de dependencia anotadas manualmente en el corpus; la precisión es el porcentaje de relaciones de dependencia encontradas por el analizador que también se encuentran entre las relaciones de dependencias anotadas manualmente en el corpus. Además, Minipar se ejecuta muy eficientemente por lo que resulta muy útil como subsistema.

La entrada de Minipar es un fichero de texto con el discurso en lenguaje natural (inglés) que se pretende analizar y su salida es un fichero de texto en el que, mediante tuplas, se indican las relaciones de dependencia entre las palabras del discurso analizado. Estas tuplas contienen la siguiente información:

La palabra considerada, su categoría léxica, el núcleo de la palabra considerada (la palabra del discurso de la que depende) y el tipo de relación de dependencia (por ejemplo: sujeto, adjunto, complemento, especificador, etcétera).

El sistema descrito en la presente memoria normaliza los fragmentos de texto (textos e hipótesis) proporcionados en el corpus para alimentar al analizador Minipar. Cuando éste ha actuado, toma los ficheros de texto que contienen las tuplas que representan el análisis y construye con esa información árboles de dependencias en memoria.

2.2. Trabajos ajenos a nuestro desarrollo

- Es un analizador estadístico, eficiente para aplicaciones exigentes, como QA, Opinion Mining de Attardi [3]. Utiliza estrategia ascendente determinista. Los analizadores estadísticos tradicionales son entrenados directamente en la selección de un árbol de análisis de una frase. En su lugar, lo entrenamos con estrategia desplazamiento/reducción (shift/reduce) para que aprenda la secuencia de acciones de análisis necesarias para generar el árbol de análisis.

- Mientras que un parser tradicional necesita una gramática para generar los árboles candidatos, un parser desplazamiento/reducción no necesita gramática alguna. Realiza el parseo mediante acciones desplaza/reduce. Aprende de un corpus anotado la acción a realizar en cada paso.

- Otra alternativa de analizador desplazamiento/reducción (shift/reduce) es la de Ming [20]. En él las acciones son seleccionadas por un clasificador, a partir de un conjunto extendido de acciones y búsqueda hacia delante (look ahead). Política de control: de izquierda a derecha, con paso atrás. Tipos de dependencias: separador multiclase. Es Multilingüe.

- Otra aproximación es la de Nivre, Hall y Nilsson [15] con SVM (Support Vector Machine). Utiliza un algoritmo determinista para construir grafos de dependencias con etiquetado descriptivo en tiempo lineal y modelos basados en la historia para realizar la predicción de la siguiente acción del analizador en los puntos de elección no deterministas.

- El método de Nivre es un algoritmo de análisis lineal en el tiempo. Yu-Chieh et al. [28] optaron por extender la eficiencia en tiempo del método de Nivre. No analizan la secuencia de palabras varias veces: utilizan el algoritmo de Nivre solamente para las palabras sin analizar. Se etiquetan los root de forma eficiente.

- Intentan reducir la tasa de tokens sin analizar mediante:

  - Análisis hacia delante.
  - Análisis hacia atrás (backwards parsing) que suele ser mejor.

- Para clasificar las palabras que quedan en la pila: Utilizan un analizador para roots (root parser) Para reconectar las palabras no descriptivas emplean un postprocesador para reconstruir los arcos, que realiza un análisis exhaustivo desde el comienzo de la frase.

2.3. Métricas de evaluación

Nos aproximamos a un foro de obligada referencia internacional como es "CoNLL-X Shared Task: Multi-lingual Dependency Parsing" [1]. Este foro está dedicado al análisis de

[1] http://nextens.uvt.nl/~conll/

	Arabic	Chinese	Czech	Danish	Dutch	German	Japanese	Portuguese	Slovene	Spanish	Swedish	Turkish	AV	SD	Bulgarian	Name	Affiliation
	57.64	78.37	60.92	77.90	74.59	77.56	87.41	77.42	59.19	68.32	79.15	51.07	70.80	11.11	78.74	Sander Canisius, Antal van den Bosch, Erik Tjong Kim Sang, Toine Bogers, Jeroen Geertzen	Tilburg University
	53.81	54.89	59.76	66.35	58.24	69.77	65.38	75.36	57.19	67.44	68.77	37.80	61.23	9.92	72.89	Giuseppe Attardi	Universita di Pisa
	63.81	74.81	59.36	78.38	68.45	76.52	90.11	81.47	67.83	72.99	71.72	55.09	71.71	9.67	79.73	YuChieh Wu	National Central University
	60.94	83.68	68.82	79.74	67.25	82.41	88.13	83.37	68.43	77.16	78.65	58.06	74.72	9.72	83.30	Xavier Carreras, Mihai Surdeanu, Lluís Màrquez	Technical University of Catalonia
	52.42	72.72	51.86	71.56	62.75	63.82	84.35	70.35	55.06	69.63	65.23	60.31	65.01	9.46	73.49	Deniz Yuret	Koc University
	55.37	76.18	63.02	74.61	69.51	74.74	84.75	78.18	64.31	71.37	74.09	53.87	70.00	9.25	79.21	Eckhard Bick	University of Southern Denmark
	66.71	86.92	78.42	84.77	78.59	85.82	91.65	87.60	70.30	81.29	84.58	65.68	80.19	8.53	87.41	Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, Svetoslav Marinov	Växjö University, Istanbul Technical University, University of Skövde

Figura 2.1: Algunos datos de CoNLL-X (LAS)

dependencias multilingüe, en él podemos encontrar múltiples resultados por diferentes grupos de trabajo, que nos marcan entorno a qué valores deben rondar los nuestros; pero sobre todo nos aporta un modelo para realizar la propia evaluación de nuestros resultados.

CoNLL-X Shared Task proporciona una serie de programas en PERL para la evaluación de los resultados. Esto nos da un estándar para la comparación de nuestros resultados con los obtenidos por otros investigadores. Las diferentes métricas de evaluación son: Labeled attachment score (LAS), Unlabelled attachment score (UAS), Unlabelled attachment score(UAS) y Label accuracy.

2.3.1. Labeled attachment score (LAS)

También denominada métrica oficial, este método calcula el porcentaje de tokens para los cuales el sistema predijo correctamente tanto su cabeza (token al que apunta) como su relación de dependencia con la misma. Este método no tiene en cuenta aquellos token que pertenecen al la categoría "Punctuation" del standard UNICODE. Como por ejemplo: "." ", " "?" "(" "¿" "..." "_" "-" "%"... Podemos ver una muestra de resultados reales, con la métrica LAS, por otros grupos de investigación en la Figura 2.1

2.3.2. Métrica para el análisis de errores y comparación del sistema

En este modelo de métrica se obtienen dos valores: el Unlabelled attachment score (UAS) y Label accuracy. Este modelo se diferencia del LAS en que los valores porcentuales, tanto de la relación de dependencia con su cabeza, como la propia obtención de la misma, son independientes. Con lo que nos da una mayor información de los resultados de la evaluación.

	Arabic	Chinese	Czech	Danish	Dutch	German	Japanese	Portuguese	Slovene	Spanish	Swedish	Turkish	AV	SD	Bulgarian	Name	Affiliation
	74.59	82.86	72.88	82.93	77.79	80.01	89.67	85.61	74.02	71.33	85.08	64.19	78.41	7.28	82.51	Sander Canisius, Antal van den Bosch, Erik Tjong Kim Sang, Toine Bogers, Jeroen Geertzen	Tilburg University
	69.50	81.33	73.44	78.84	68.93	80.25	82.05	85.03	72.14	74.25	83.03	65.25	76.17	6.42	85.24	Giuseppe Attardi	Universita di Pisa
	75.45	79.48	74.82	83.39	71.75	79.73	91.74	85.57	76.92	76.20	76.24	69.25	78.38	6.17	85.50	YuChieh Wu	National Central University
	72.65	88.65	77.44	85.67	71.39	85.90	90.79	87.76	77.72	80.77	85.54	70.05	81.19	7.21	88.81	Xavier Carreras, Mihai Surdeanu, Lluís Màrquez	Technical University of Catalonia
	68.82	78.37	66.36	78.16	66.17	67.71	87.31	79.46	70.60	73.89	73.25	71.54	73.47	6.36	78.56	Deniz Yuret	Koc University
	68.98	83.06	72.24	80.54	74.47	79.79	87.85	84.29	75.06	75.76	82.65	65.50	77.52	6.66	84.16	Eckhard Bick	University of Southern Denmark
	77.52	90.54	84.80	89.80	81.35	88.76	93.10	91.22	78.72	84.67	89.50	75.82	85.48	5.90	91.72	Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, Svetoslav Marinov	Växjö University, Istanbul Technical University, University of Skövde

Figura 2.2: Algunos datos de CoNLL-X (UAS)

2.3.3. Unlabelled attachment score (UAS)

Este método calcula el porcentaje de tokens para los cuales el sistema predijo correctamente su cabeza. En la siguente tabla mostramos algunos de los resultados que se pueden ver en la Web de "CoNLL-X Shared Task". Podemos ver una muestra de resultados reales, con la métrica UAS, por otros grupos de investigación en la Figura 2.2

2.3.4. Label accuracy

Este método calcula el porcentaje de tokens para los cuales el sistema predijo correctamete su relación de dependencia. Podemos ver una muestra de resultados reales, con la métrica Label accuracy, por otros grupos de investigación en la Figura 2.3

	Arabic	Chinese	Czech	Danish	Dutch	German	Japanese	Portuguese	Slovene	Spanish	Swedish	Turkish	AV	SD	Bulgarian	Name	Affiliation
	74.59	82.86	72.88	82.93	77.79	80.01	89.67	85.61	74.02	71.33	85.08	64.19	78.41	7.28	82.51	Sander Canisius, Antal van den Bosch, Erik Tjong Kim Sang, Toine Bogers, Jeroen Geertzen	Tilburg University
	69.50	81.33	73.44	78.84	68.93	80.25	82.05	85.03	72.14	74.25	83.03	65.25	76.17	6.42	85.24	Giuseppe Attardi	Universita di Pisa
	75.45	79.48	74.82	83.39	71.75	79.73	91.74	85.57	76.92	76.20	76.24	69.25	78.38	6.17	85.50	YuChieh Wu	National Central University
	72.65	88.65	77.44	85.67	71.39	85.90	90.79	87.76	77.72	80.77	85.54	70.05	81.19	7.21	88.81	Xavier Carreras, Mihai Surdeanu, Lluís Màrquez	Technical University of Catalonia
	68.82	78.37	66.36	78.16	66.17	67.71	87.31	79.46	70.60	73.89	73.25	71.54	73.47	6.36	78.56	Deniz Yuret	Koc University
	68.98	83.06	72.24	80.54	74.47	79.79	87.85	84.29	75.06	75.76	82.65	65.50	77.52	6.66	84.16	Eckhard Bick	University of Southern Denmark
	77.52	90.54	84.80	89.80	81.35	88.76	93.10	91.22	78.72	84.67	89.50	75.82	85.48	5.90	91.72	Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, Svetoslav Marinov	Växjö University, Istanbul Technical University, University of Skövde

Figura 2.3: Algunos datos de CoNLL-X (UAS)

Capítulo 3

Recursos utilizados

A la hora de la implementación, se plantearon bastantes objetivos. Para la representación del análisis realizado, crear una herramienta que dibujara los árboles sería engorroso. Para el análisis de dependencias en sí, crear unos algoritmos propios e implementarlos, para crear un sistema experto de análisis, es una tarea, para la que en otros lugares (Växjo, Suecia; Standford, EE UU; Manitoba, Canadá), se dedican equipos enteros de investigación. Para el entrenamiento de la aplicación, crear un corpus, lo suficientemente grande para abarcar todos los posibles casos sería igualmente trabajoso. Por el mismo motivo que para el análisis, para el etiquetado del Part-of-Speech la generación de un algoritmo de etiquetado, sería motivo de un proyecto de SI aparte.

Para resolver estos problemas de "ambición", decidimos acudir al uso de aplicaciones de terceros. Ello implica una dependencia grande de ellas, pero se estudió y se llegó a la conclusión de que merecía la pena. A continuación se explican detalladamente cada uno de los recursos utilizados.

3.1. El corpus Cast3LB

Actualmente, el corpus Cast3LB está formado por 100.000 palabras aproximadamente. Fue desarrollado por el *Grupo de investigación en Procesamiento del Lenguaje y Sistemas de Información* de la Universidad de Alicante [21]. A nivel sintáctico, en el corpus Cast3LB se han anotado los constituyentes sintácticos (oraciones, sintagmas, etc.) y las relaciones funcionales básicas (sujeto, objeto directo, etc.).

A nivel semántico se ha anotado el sentido de cada verbo, nombre y adjetivo. Para la representación del sentido se ha utilizado el número de sentido del WordNet de cada lengua. Esta representación del sentido es la misma para las tres lenguas, dado que es el número del Interlingua Index de EuroWordNet. A nivel pragmático-textual se están anotando las principales anáforas (sujetos elípticos, pronombres personales y clíticos, etc.) y su antecedente.

El corpus Cast3LB está implementado mediante archivos de unas 15 oraciones aproximadamente, en formato XML, para poder etiquetar y darle la mayor riqueza posible a la oración. Las oraciones están analizadas en constituyentes, que no dependencias, por

lo que más tarde se implementará el algoritmo de Gelbukh para pasar de análisis de constituyentes, al análisis de dependencias. Ahora pasamos a explicar el contenido de los archivos XML.

De manera ordenada y progresiva iremos poniendo parte del código y lo analizaremos y explicaremos.

3.1.1. Cabecera y etiquetas FILE y LOG de los archivos del corpus Cast3LB

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FILE SYSTEM "3lb.dtd">
<FILE id="agset" language="es" wn="1.5" ewn="dic2002" parsing_state="process"
semantic_state="process" last_modified="13-01-2006" project="3LB" about="3LB
project annotation file">
<LOG auto_file="a1-0-auto3.log" anno_file="a1-0-anno4.log"
nosense_file="a1-0-nosense4.log" />
```

Figura 3.1: Cabecera y etiquetas FILE y LOG de los archivos del corpus

En cuanto a la etiqueta <FILE>hay poco que decir. *Id* es el identificador del fichero, *language* es el idioma, *wn* se refiere a la versión del wordnet, *ewn* se refiere a la edición diccionario del wordnet utilizada, *parsing_state* y *semantic_state* se refieren al estado del análisis y semántica respectivamente, *last_modified* anota la última modificación realizada, *project* el nombre del probecto y *about* sobre lo que trata éste.

Lo mismo para la etiqueta <LOG>que indica los ficheros en los que de alguna forma se irá almacenando información referente al proceso. En la figura 3.1, mostramos una cabecera típica.

3.1.2. Estructura de las frases

A continuación vamos a trabajar sobre una sola frase; la primera del archivo *a1-0.xml*. La frase en cuestión es: *"Medardo_Fraile juega a un cinismo fácil y divertido.*".

La etiqueta <SENTENCE>identifica la frase que trataremos dentro del fichero, pues hay numerosas. Con la etiqueta <Anchor>se identifica cada palabra de la frase y el ancho de la misma o también se puede ver como la posición en la sentencia. En la figura 3.2 se muestra un ejemplo de código.

Con respecto al cuadro anterior vemos que la primera palabra (*Medardo_Fraile*) empieza en la posición 0 y como tiene 14 letras / símbolos la siguiente palabra (*juega*) estará a partir de la posición 15 pues entre ambas hay que situar un espacio. Así continuaríamos secuencialmente hasta el punto que también se tiene en cuenta como todos los diversos signos de puntuación.

3.1.3. Etiquetas Annotation

Para cada palabra de la frase o sentencia nos encontramos con un conjunto de anotaciones de índole diversa que se implementan mediante las etiquetas *Annotation*.

```
<SENTENCE id="agset_1">
<Anchor id="agset_1_ac1" offset="0"/>
<Anchor id="agset_1_ac2" offset="15"/>
<Anchor id="agset_1_ac3" offset="21"/>
<Anchor id="agset_1_ac4" offset="23"/>
<Anchor id="agset_1_ac5" offset="26"/>
<Anchor id="agset_1_ac6" offset="34"/>
<Anchor id="agset_1_ac7" offset="40"/>
<Anchor id="agset_1_ac8" offset="42"/>
<Anchor id="agset_1_ac9" offset="52"/>
<Anchor id="agset_1_ac10" offset="54"/>
```

Figura 3.2: Desglose de la etiqueta Sentence

```
<Annotation id="agset_1_an3" start="agset_1_ac1" end="agset_1_ac2" type="syn">
<Feature name="roles">SUJ</Feature><Feature name="label">sn</Feature>
<Feature name="parent">agset_1_an2</Feature></Annotation>
```

Figura 3.3: Desglose de la etiqueta Annotation

Las etiquetas *Annotation* las hay de cuatro tipos *"syn"* referente a las sintaxis, *"wrd"* referente a la palabra en sí, *"pos"* referente a la posición de la palabra y *"dummy_root"* que indica la raíz. Además se le asigna un identificador (*id ="agset_1_an3"*) que se utilizará para referenciar esta etiqueta a modo de árbol. El numero de etiquetas Annotation que se le asignan a una palabra varía según la misma, pero podemos diferenciar cuales corresponden a cada una observando los campos *"start"* y *"end"* que utilizan los identificadores creados dentro de la sección Anchor. Dentro de Annotation se definen otras etiquetas de tipo Feature que hacen referencia al rol (*"roles"*), lo que se define (*"label"*), A quién apunta o de quién es hijo (*"parent"*), una denominación de la palabra (*"sense"*) o de quién procede (*"lema"*).

3.1.4. Análisis de la oración

Tomando el identificador de la etiqueta Annotation y fijándonos en el valor del atributo "*parent*" de la etiqueta *Feature*, siendo este último el padre del identificador, podemos ir formando nodos que constituyan el árbol de análisis. Una vez obtenido el árbol si, manteniendo la estructura, nos quedamos solo con los sintagmas y las palabras de la oración que se desprenden de S el resultado será el que se aprecia en la figura 3.4. Por otro lado en la figura 3.5 podemos ver, sobre la frase, el resultado del análisis desde el punto de vista de los sintagmas y de la funcionalidad de los roles. Es de vital importancia ser rigurosos tanto con el orden de formación como con los niveles del árbol, pues determinarán cómo se engloban o a quién pertenecen las palabras o sintagmas.

3.2. Treetagger

Treetagger [1] es un herramienta para la anotación de texto con part-of-speech (categorías gramaticales) que ha sido desarrollada por TC project - Textcorpora und Er-

[1] http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/

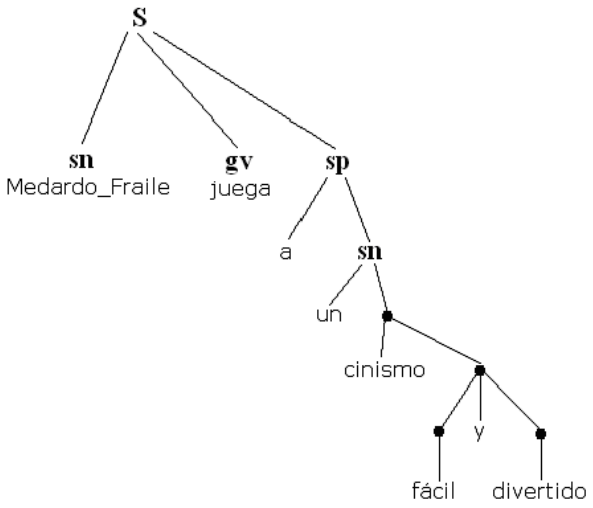

Figura 3.4: Análisis de la oración del ejemplo

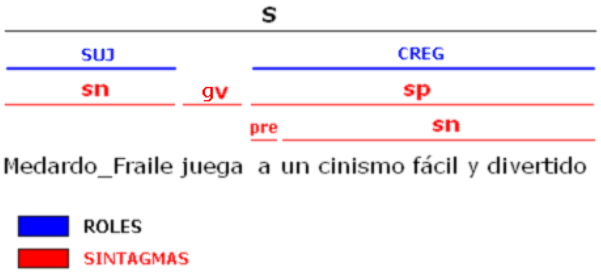

Figura 3.5: Análisis de la oración del ejemplo

shliessungswerkzeug ('textual corpora and tools for ther expliration') en la Universidad de Stuttgart con la participación de los siguientes organismos e instituciones: Institute for Romance Linguistics, Institute for Computer Science (department of artificial intelligence). Aunque en nuestro caso hemos utilizado la versión para el Castellano también están disponibles las versiones para Alemán, Inglés, Francés, Italiano, Holandés, Búlgaro, Ruso, Portugués y Griego

En 1993/1994 el proyecto recogió material textual para el Alemán, Francés e Italiano, desarrollando una representación para textos computando un conjunto de etiquetas asignadas a elementos de un texto indicando sus relaciones lógicas o estructurales con el resto de el texto, junto con un lenguaje de consultas y un sistema de acceso al corpus para la exploración lingüística de los textos que forman parte del material. Los resultados de los textos y análisis son mantenidos y separados, por razones de flexibilidad y capacidad de extensión del sistema. Esto es posible debido a un particular aprovechamiento del almacenamiento y representación.

3.2.1. Motivación de su uso en nuestra aplicación

Una de las funcionalidades que nos interesaba introducir en nuestra aplicación era que un usuario, al que no tiene porqué interesarle el entrenamiento y evaluación de corpus de textos, pudiera realizar estudios pormenorizados de las oraciones que decidiera, sin la necesidad de realizar un etiquetado previo. Esto conlleva que un usuario introduzca simplemente una frase o texto y JBeaver lo analice y construya los árboles. Para eso nuestro programa debe realizar un estudio morfológico con el correspondiente etiquetado de las categorías gramaticales (part-of-speach). Esta es la razón que nos llevó a la búsqueda y posterior implantación del treetagger.

Para mostrar un ejemplo de este proceso adjuntamos a continuación unas capturas de pantalla (Figuras 3.6 y 3.7) que muestran mejor la funcionalidad comentada.

En la Figura 3.6, como vemos en el título de la ventana, el usuario ha cargado el modo "Parsing", esto se hace por medio del menú de la barra Tools->View. Por esta razón le aparece un área de texto donde introducir el texto a analizar. Posteriormente sólo tiene que pulsar la tecla START. En la figura 3.7 el programa nos muestra un desplegable (1)

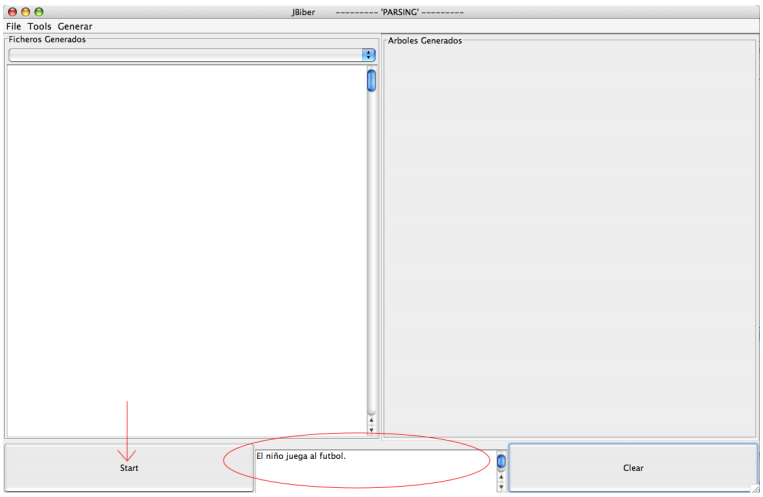

Figura 3.6: Introducción de una oración para su análisis

con todas las oraciones del texto que en este caso solo hay una. En otra área de texto (2) podemos ver las oraciones ya analizadas y en formato *.tab. Tras seleccionar en en el desplegable la oración a visualizar solo tenemos que ir al menú de la barra Generar para obtener el gráfico (3).

3.2.2. Proceso de implantación.

Como ya se ha comentado en secciones anteriores a esta, el formato de corpus de entrada es el Cast3LB. Dicho corpus es usado para los diferentes entrenamientos de la aplicación y para la creación de otro corpus de dependencias evaluado y útil para las pruebas y evaluaciones por parte de los usuarios. De ello se desprende que el formato de categorías gramaticales que usa no es compatible con Treetagger, lo que provoca que ya no solo cuando el usuario introduce frases o textos, sino también siempre que queremos

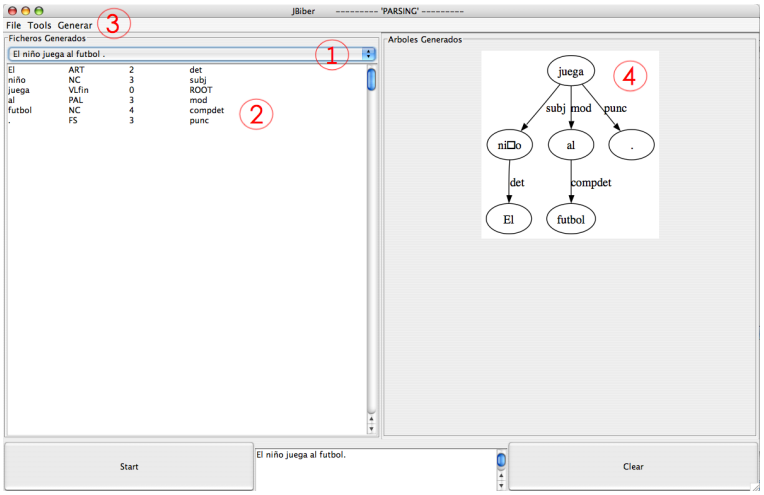

Figura 3.7: Ventana de visualización de los resultados

realizar entrenamientos con formato Cast3LB, tengamos que realizar una conversión de las etiquetas para entrenar con estas últimas al propio Maltparser.

Existen diversos factores a tener en cuenta en este proceso de transformación.

- Las conjuntos de etiquetas de las categorías gramaticales no son, en su mayor parte, equivalentes. Esto significa que para el conjunto de palabras que el Cast3LB etiqueta con un tag el Treetagger puede diferenciar entre varias y viceversa, incluso que no existan en alguno de los corpus. Como la transformación de árboles de constituyentes en árboles de dependencias se hace usando las etiquetas del Cast3LB (por tener una mayor precisión), el problema está en el caso de que una etiqueta de éste se corresponda con varias del treetagger eligiendo esta última de forma aleatoria entre todas, por la razón que se explica a continuación.

- Hay que realizar un entrenamiento con todas las etiquetas, pues si un usuario introdujera un texto que el Treetagger etiquetara con tags que no fueron usados en el proceso de entrenamiento, las consecuencias en la formación de los árboles, son del todo impredecibles por parte del Maltparser.

3.2.3. Alineamiento del etiquetado gramatical

Cast3LB	Treetagger
w	NMON
aoXXXX	ORD
aqXXXXX	ADJ
cc	CC,CCAD,CNEG,CC
cs	CSUBF,CSUBI,CSUBX,CSUBF
ddXXXX	DM
daXXXX	ART
rg	ADV
deXXXX,diXXXX	INT
vag0000,vmg0000,vsg0000	VHger,VLgerVEger,VSger,VLger
van0000,vmn0000,vsn0000	VHinf,VLinf,VMinf,VEinf,VSinf,VLinf
vap00XX,vmp00XX,vsp00XX	VHadj,VLadj,VM,VE,VS,VLinf
vXXXXXX	VHfin,VLfin,VMfin,VEfin,VSfin,VLinf
spcms	PDEL,PAL,PDEL
ncXXXXX	NC
npXXXXX	NP
pr0cn	CQUE
prXXX	REL
p000XXXX	SE
p0XXXXXX, ppXXXXXX	PPX
dpXXXX, pxXXXXX	PPO
sps00	PREP
i	ITJN
rn	NEG
dn0cp0	CARD
diXXXX	QU
Fat,Fit,Fia,Fx,Fs,Fp,Fs	FS
Faa	ALFS
Fc	CM
Fd	COLON
Fe	QT
Fh	SLASH
Fpa,Fpt	DASH
Fz	CODE
X	PE
Zm,Zm,Zp	CARD
	PNC (No encontrada)

En la anterior tabla se muestra la equivalencia entre etiquetas del Cast3LB y Treetagger. Como se puede ver, esta relación no es unívoca, por lo que de manera interna, con vistas al entrenamiento, cuando el conjunto de una categoría gramatical del Cast3LB es

menor que el del Treetagger, se elige una entre el conjunto de éstas últimas de manera probabilística.

3.3. Maltparser

MaltParser [22] es un sistema de análisis sintáctico de dependencias guiado por los datos que pretende ser independiente del idioma. Está dirigido por el profesor Joakim Nivre de la Universidad de Växjö, en Suecia[2].

Su creación está motivada por el hecho de que los parsers que están demasiado ajustados a un cierto idioma funcionan mal para otros. También busca una solución para el indeterminismo de los analizadores de dependencias. He aquí uno de los motivos por los que hemos escogido MaltParser. Al ser "genérico", es decir, para cualquier idioma, lo podemos entrenar con un corpus respetable y así poder automatizar el análisis de dependencias en castellano con elevada tasa de aciertos, que era uno de los objetivos del proyecto.

3.3.1. Modos de funcionamiento

MaltParser es un sistema que, como ya se ha comentado, está basado en el aprendizaje. Esto es, mediante un entrenamiento considerable en un cierto idioma se puede llegar a alcanzar un nivel de acierto alto. En este sentido, es aplicable a cualquier lengua, y siendo así, podemos crear tantos analizadores de dependencias como entrenamientos hagamos, sin necesidad de destruir un entrenamiento para crear uno nuevo en otro idioma. Al entrenar Maltparser, éste crea un archivo llamado modelo, mediante el cual, usado en la fase de análisis, podremos realizar nuestro análisis a partir de un texto de entrada. Una de las características que proporcionamos con JBeaver es la posibilidad de meter un texto tanto por teclado como por archivo, y de esta manera realizar el análisis para su posterior uso. De esta manera, si entrenamos MaltParser con un Corpus para el castellano generamos un modelo entrenado a partir de ese corpus. Si a continuación lo entrenamos con un corpus para el griego y generamos el modelo entrenado para el griego, cuando queramos analizar un texto en griego cogemos el modelo del griego y ya está. Si a continuación queremos compararlo con su traducción al castellano, analizamos su traducción con el modelo de castellano y tenemos los dos análisis en archivos distintos. La segunda posibilidad de MaltParser es el análisis donde somos capaces de usar un modelo para analizar sintácticamente las dependencias de oraciones en dicho idioma.

MaltParser posee unos formatos propios para trabajar. Éstos se explican más adelante, en la sección 3.3.2 en la página 28. Tanto en modo aprendizaje como en análisis, a MaltParser se le indicará todo mediante un archivo de opciones. Este archivo contendrá toda la información necesaria para poder realizar lo deseado. A continuación se explica, en cada subsección, lo necesario para cada modo.

[2]Más información en `http://w3.msi.vxu.se/~nivre/research/MaltParser.html`

3.3.1.1. Modo aprendizaje (LEARNER)

Para el modo de aprendizaje hay que indicar a través del archivo de opciones:

1. El fichero de entrada: debe estar escrito en un determinado formato: Malt-TAB o CoNLL-X shared task format. Más adelante, en la sección 3.3.2 en la página siguiente se explican con más detalle. El formato de Malt-TAB es el más sencillo. Consiste en dividir el fichero en cuatro columnas a través de tabuladores. En la primera columna se especifican las palabras, en la segunda los tokens de part-of-speech (las categorías léxicas), en la tercera el índice de la palabra cabeza (el índice 0 es el elemento destacado root) y en la cuarta la etiqueta de la dependencia.

2. Fichero de salida. Al igual que el de entrada, existe la posibilidad de especificar el formato de salida, Malt-XML, Malt-TAB, CoNLL-X shared task format, Malt-XML, CoNLL-X shared task format (XML version) y TIGER-XML. El formato Malt-XML es muy similar. Es igual a Malt-TAB pero con etiquetas xml.

3. Los formatos de los archivos tanto de entrada como de salida. Los ya citados.

4. Ficheros con el conjunto de etiquetas tanto de part-of-speech (categoría gramatical) como de dependencias. Las etiquetas part-of-speech consisten en especificar los tipos de palabras que se pueden proporcionar en entrenamiento. Las etiquetas de dependencias se refieren a los tipos de dependencias que se podrán encontrar a la hora de entrenar y son con las que se etiquetarán las dependencias de los archivos analizados.

5. El fichero que especifica las características del modelo que queremos sacar, es decir el modelo.

6. El tipo de aprendizaje junto con sus opciones. Existen dos tipos de aprendizaje. El aprendizaje basado en memoria (TiMBL) (Daelemans y Van den Bosch 2005) que almacena todas las instancias en tiempo de aprendizaje y usa ciertas variantes de la clasificación vecino k-más-cercano para predecir la siguiente acción en tiempo de análisis. MaltParser usa el paquete TiMBL para implementar este algoritmo de aprendizaje y soporta todas las opciones provistas por el paquete. El otro tipo de aprendizaje es el *Support vector machines* que depende de las funciones de núcleo para inducir el clasificador hiperplano de máximo margen en tiempo de aprendizaje, que puede ser usado para predecir la próxima acción en tiempo de análisis. MaltParser usa la librería LIBSVM (Chang y Lin 2005) para implementar este algoritmo de aprendizaje y soporta todas las opciones provistas por el paquete.

3.3.1.2. Modo Análisis (PARSER)

Al igual que para el modo aprendizaje o entrenamiento se proporciona a MaltParser todo lo necesario mediante un archivo de opciones. A continuación se explica lo necesario a incluir en este archivo:

1. El fichero de entrada: Un fichero similar al de modo LEARNER pero en este caso simplemente hay que indicar la palabra junto con su análisis léxico. Esto es, sólo la primera y la segunda columna. Como se ha dicho antes, en la sección 3.3.2 se explica con más detalle.

2. El fichero de salida.

3. Los formatos de los archivos, tanto de entrada como de salida. Igual que para el aprendizaje.

4. Los ficheros que especifican las etiquetas léxicas del part-of-speech y de dependencias.

5. El fichero que especifica el modelo entrenado en el que nos basaremos para realizar el análisis. Evidentemente, ha de ser un modelo entrenado con las etiquetas del punto 4, y en el idioma para el que queremos realizar el análisis.

6. El algoritmo de análisis junto con sus opciones. Existen dos posibles algoritmos para el análisis. El primero de ellos, y el usado por defecto es el de Nivre (2003, 2004). Este algoritmo es de complejidad lineal para estructuras limitadas de dependencias. Puede ser ejecutado en modo *arco-ambicioso* (-a E) o en modo *arco-standard* (-a S)(cfr. Nivre 2004). El otro algoritmo posible es el de Covington (2001). Este algoritmo es de complejidad cuadrática para estructuras irrestrictas de dependencias, que procede probando a encadenar cada nuevo token con el token anterior. Puede ser usado en modo *potencial* (-g P) , donde la operación de encadenado está restringida a la estructura potencial de dependencia, o en un modo *no-potencial* (-g N), permitiendo estructuras de dependencias no-potenciales (pero acíclicas).

A su vez el modo PARSER devuelve un archivo (formato Malt-XML o Malt-TAB, en función de las opciones especificadas), con un análisis de dependencias realizado según el modelo.

3.3.2. Formatos de MaltParser

MaltParser admite como entrada dos formatos. El primero es MaltTAB y el segundo es el formato de trabajo compartido de CoNLL-X (CoNLL-X shared task format). Nosotros nos centraremos en la explicación del formato MaltTAB ya que es el formato que hemos usado y que pensamos que es más potente.

El formato MaltTAB consiste en dividir la oración u oraciones en lineas y columnas. A su vez dentro del formato MaltTAB hay dos variantes, la de aprendizaje y la de análisis. Cada línea corresponderá a una palabra de la oración. El orden es el que aparece en la oración. Para cada línea el contenido de las otras columnas variará en función de su objetivo, análisis o aprendizaje. Cada palabra tiene cinco atributos:

1. id = Identificador único dentro de la frase. (requerido)

2. form = Forma léxica de la palabra (String). (requerido)

```
Hay vaip3s0 0 ROOT
veces ncfp000 1 subj
que pr0cn000 7 whn
el da0ms0 5 det
cuerpo ncms000 7 subj
nos pp1cp000 7 DEF
engaña vmip3s0 2 vrel
por sps00 7 mod
el da0ms0 11 det
puro aq0ms0 11 pnmod
placer ncms000 8 compdet
de sps00 11 mod
engañarnos vmn0000 12 DEF
. Fp 1 punc
```

Figura 3.8: Ejemplo de formato MaltTAB

3. postag = Etiqueta del part-of-speech o categoría léxica. (opcional)

4. head = Cabeza sintáctica (identificador de palabra). (opcional)

5. deprel = Relación de dependencia con la cabeza. (opcional)

El único atributo requerido es el form. El id es implícito. El "etiquetado" de la oración «Hay veces que el cuerpo nos engaña por el puro placer de engañarnos.» quedaría como se ve en la figura 3.8. La cabeza del ROOT es evidentemente el 0 porque no tiene regente. Este ejemplo sería el formato de entrada para el método de aprendizaje o el formato de salida para los dos modos.

El formato Malt-XML es similar. Para cada oración hay una subcategoría palabra, y cada una de estas contienen los datos en forma de etiquetas.

3.4. GraphViz

GraphViz es la solución que encontramos para poder dibujar los árboles resultantes del análisis. Barajamos varias opciones, pero al final nos decantamos por GraphViz por su potencial y buen funcionamiento. *dot* es, por decirlo de alguna manera, el compilador de GraphViz. Es una aplicación disponible para las tres grandes plataformas: Linux, MacOSX y Windows XP.

La forma de trabajo, y donde nosotros vimos el potencial y la facilidad de su uso, fue en que *dot* dibuja grafos dirigidos a partir de un archivo de texto plano, escrito en lenguaje DOT. La invocación de la aplicación es mediante línea de comandos y el archivo de salida es decidida por el usuario, dentro de unos formatos disponibles: GIF, PNG, SVG o PostScript (que puede ser fácilmente convertido en PDF).

En la figura 3.9 se aprecia un ejemplo del contenido de un archivo .dot. Su contenido es el del análisis de dependencias de una oración bastante sencilla. El resultado de la siguiente invocación sería el de la figura 3.10.

Como se puede observar, la construcción de un grafo dirigido es relativamente sencilla. El nodo origen se coloca a la izquierda y mediante una flecha (->) se pone a la derecha

```
digraph 0 {
es ->perro[label="SUJ"];
es ->marron[label="ADV"];
es ->pto[label="PTO"];
perro ->El[label="DET"];
}
```

Figura 3.9: Ejemplo de código en lenguaje DOT.

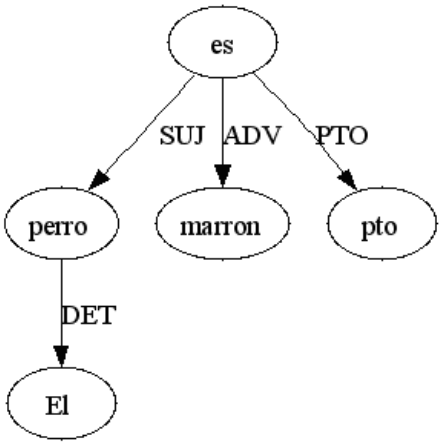

Figura 3.10: Grafo resultante del código de la figura 3.9.

el hijo. Si posee varios nodos dependientes se escribe cada una de las aristas. Para que cada arista tenga una etiqueta, se coloca a lado de la conexión `padre ->hijo` la *label* correspondiente. En el ejemplo, `es ->perro[label="SUJ"];`tiene la *label* de `"SUJ"` porque `perro` ejerce de sujeto en la oración. Cada nodo tiene un identificador. En el ejemplo, el identificador y el label del nodo coinciden, porque no se dice lo contrario. Existe la posibilidad de que teniendo un identificador para la conexión de los nodos en su label aparezca *algo* distinto. Por ejemplo, para `es ->perro[label="SUJ"];` si colocamos antes una linea tal que `perro[label="PULGOSO"];` en el dibujo aparecería dentro del nodo referente a `perro` su etiqueta, es decir, `PULGOSO`.

En el artículo *Drawing graphs with dot* de Emden Gansner et al. se puede profundizar en el uso de esta potente herramienta. Nosotros simplemente decidimos usar lo justo para poder representar los árboles de dependencias sin demasiada complicación, ya que, aunque *dot* es capaz de mucho, la sobrecarga de trabajo para generar grafos con más detalle, podría suponer un rebaja en el rendimiento.

Capítulo 4

Proceso de desarrollo de JBeaver

JBeaver es un sistema de análisis y generación de analizadores de dependencias. Podemos crear corpus de entrenamiento, entrenar a un sistema automático de aprendizaje y por último realizar análisis y evaluarlos tanto gráfica como estadísticamente. Todo ello siendo éste un sistema autónomo, fácil de usar, portable, con un alto rendimiento y por supuesto, debido a su carácter didáctico y de investigación, público y gratuito.

Puesto que todo el trabajo parte de cero, el proceso de desarrollo ha sido largo y duro. Primeramente queríamos crear el corpus de dependencias para lo que nos pusimos en contacto con la Universidad de Alicante que nos cedieron su corpus de anotación sintáctica Cast3LB. Pero este corpus utiliza el paradigma de constituyentes por lo que tuvimos que crear un algoritmo de transformación de árboles de constituyentes en árboles de dependencias, este algoritmo no es para nada trivial y requiere de una gran capacidad creativa. Nos basamos en un algoritmo teórico propuesto por Gelbukh de la Universidad Autónoma de México, ampliándolo y mejorándolo, pues como se ha dicho es un algoritmo que no tenemos constancia de que se haya implementado antes.

Una vez ya tenemos nuestro corpus, entrenamos la herramienta de aprendizaje automático Maltparser_0.4, que nos otorga una gran versatilidad pudiendo parametrizar multitud de opciones, como por ejemplo elegir el propio algoritmo de aprendizaje entre Covington y Nivre. Una vez entrenado este modelo, había que probarlo analizando diferentes clases de textos (medicina, economía, Deportes...) pero puesto que las reglas de nuestro algoritmo de marcado de dependencias se basan en el etiquetado del corpus, necesitamos procesar los textos añadiéndole sus part-of-speach (categorías gramaticales: artículo, verbo, determinante...). Utilizamos la herramienta Treetagger, pero el etiquetado del corpus Cast3LB no es el mismo y ni siquiera unívoco con las etiquetas del Treetagger; de ahí que realizáramos un algaoritmo para su alineamiento. Esto es de vital importancia, pues si no se hace muy finamente puede provocar que el Maltparser_0.4 no analice los textos correctamente.

Para terminar comentar que añadimos una última funcionalidad a nuestra aplicación para que ésta pudiera evaluar los resultados de los análisis cuantitativa y cualitativamente usando las métricas propuestas por el foro CoNLL-X Shared Task: Label attachment score (LAS), Unlabel attachment score (UAS) y Label accuracy. Decir también que

todos los árboles se pueden ver gráfica y dinámicamente a través de nuestro sistema para su evaluación visual, esto nos permite acercarnos a lingüistas expertos para buscar su colaboración pues como ya se ha comentado el proceso de marcado de dependencias es creativo y requiera de una cierta preparación.

4.1. Recursos y requisistos

La realización de la aplicación JBeaver tiene los siguientes requisitos:

- Sistemas Operativos: Linux, Mac OS X
- Versión 5.0 de la máquina virtual Java.
- Instalación previa del Graphviz. (para la utilización de la sub-herramienta Graph-Dot)
- Los sistemas Treetagger y MaltParser_0.4 ya vienen incluídos en el paquete del proyecto.

Decidimos la utilización de la plataforma Java por tener todos los componentes del grupo un basto conocimiento de la misma y pues nos daba una mayor versatilidad a la hora de que funcionara en todas las plataformas operativas; como nuestro proyecto es de clara divulgación cientíica creímos oportuno que esto fuera así. De ahí que trabajáramos en diferentes sistemas operativos teniendo las primeras vesiones para aquellas que se indican arriba. No tenemos versión para otras plataformas, debido a la dependencia que tiene nuestra aplicación de alguno de los software nombrados y que o bien no tiene una versión para ellas o bien su instalación e invocación es compleja.

Enlazando con esto último nos da los argumentos necesarios que nos han llevado a ... por los software y no otros. Po ejemplo para el etiquetado de categorías léxicas se barajaron diversas aplicacones como Treetagger, Freeling etc y la complejidad a la hora de instalar algunas como en concreto esta última (freeling) nos llevó a decidirnos por Treetagger. De igual forma pasón con el programa de creación de gráficos, decidiéndonos por el uso del Graphviz gracias a su facilidad de uso instalación y multitud de versiones para todas las plataformas.

4.2. Ciclo de implementación

En el desarrollo hemos seguido un proceso iterativo dividido en las diferentes fases que se muestran en la figura 4.1. Debido en parte a la conplejidad de la aplicación y a la posibilidad de mejoras y ampliaciones que preveíamos irían surgiendo, nos fue posible hace una planificación detallada, asignando a cada persona del grupo una tarea para cada día del curso. De ahí que dividimos el proceso iterativo en dos planes:

- Plan de fase: En este plan asociamos los grandes hitos del proyecto a las fechas importantes del curso, a saber: Navidades, comienzo de exámenes de Febrero, Se-

mana Santa y comienzo de exámenes de Junio, que ratíficamos con las pertinentes reuniones.

- Plan de iteraciones: El plan de iteraciones marcado viene caracterizado por una gran inversión de tiempo en la investigación de las herramientas en las que se basa nuestro proyecto, buscar las tecnolgias adecadas para llevar a cabo nuestros objetivos y la búsqueda y creación de los algoritmos que solucionase el problema.

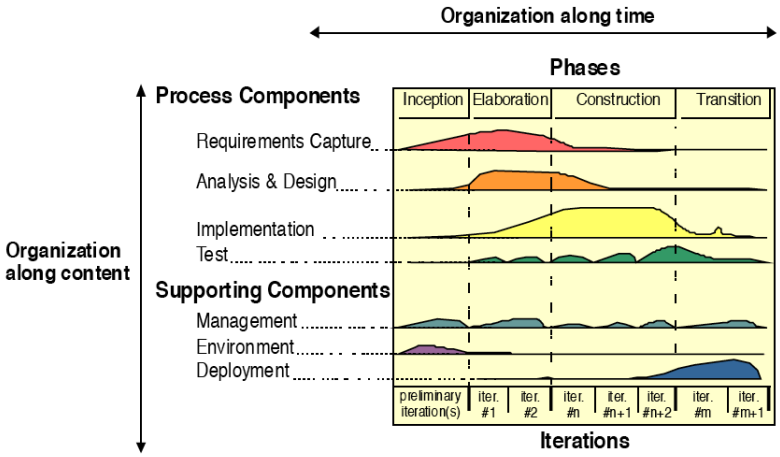

Figura 4.1:

Es importante dejar claro en el proceso de desarrollo lo referente al desarrollo del corpous de entrenamiento y evaluación y por otro lo concerniente al desarrollo del software.

4.3. Desarrollo del software

El proceso de desarrollo del software ha estado dividido en los diferentes módulos del sistema, que se repartían la funcionalidad de la aplicación. Por un lado, la creación del corpus que nos serviría para el posterior entrenamiento de la herramienta de aprendizaje automático *MaltParser_0.4*. Seguidamente todo lo referente a la integración en nuestro sistema de la herramienta *Maltparser* y posterior entrenamiento de la misma, con el corpus de dependencias ya creado. Una vez ya tuvimos la herramienta integrada se creó el módulo de anális que como su propio nombre indica analiza los textos gracias al modelo generado en el entrenamiento. Para terminar se crearon dos módulos: El de evaluación, que nos dice la precisión de nuestros resultados y el de visualización que nos muestra gráficamente los árboles de dependencias para su estudio.

Sin ánimo de ser exhaustivos la estructura del código de la aplicación tiene la siguiente forma (Figura 4.2)

Nos parece que la parte del código más interesante, es lo que concierne a la implementación del código teórico de Gelbukh, por lo que mostramos con otro diagrama de UML como las clases se relacionan entre sí (Figura 4.3).

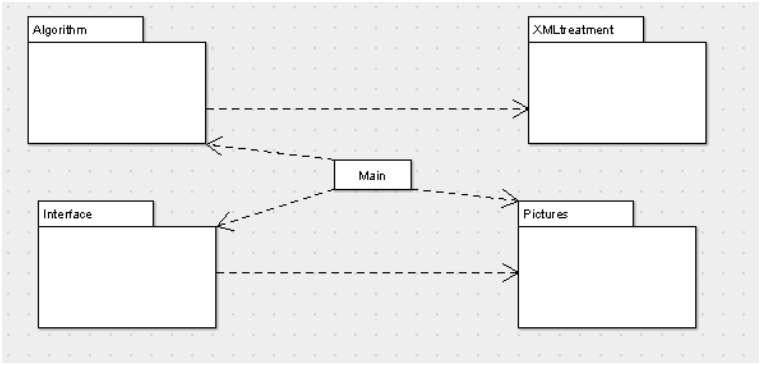

Figura 4.2: Diagrama de clases (Paquetes)

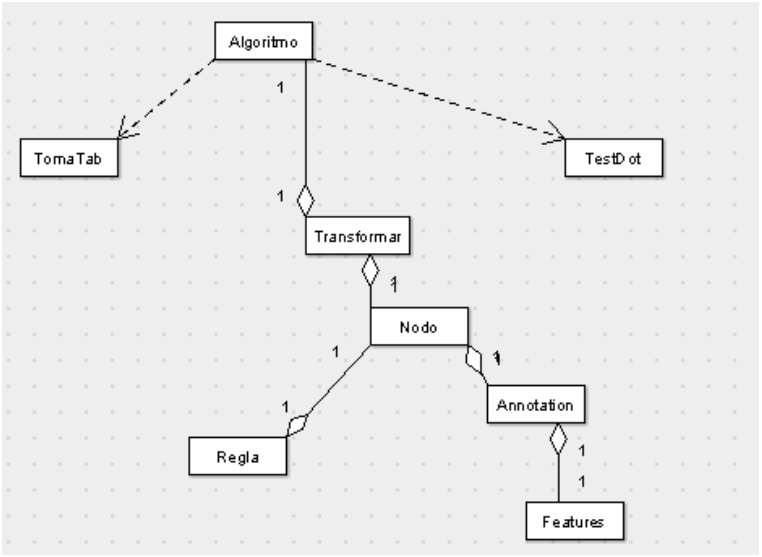

Figura 4.3: Diagrama de clases.

4.3.1. Desarrollo del Corpus

La creación del Corpus de dependencias fue una tarea árdua y profunda pues requiere del estudio e investigación tanto de los algoritmos de transformación de constituyentes en dependencias como de los propios formatos de entrada, además del procesado de los mismos; transformándolos en estructuras internas de nuestro sistema.

4.3.1.1. Traducción de los archivos XML de Cast3LB a la estructura interna de árboles de constituyentes

Como hemos ya hemos mencionado, el Corpus Cast3LB está etiquetado en el formalismo del análisis de constituyentes (de estructura de frase). Nosotros queremos tener un corpus de entrenamiento etiquetado en el formalismo de dependencias. Este fué el primero de los problemas que tuvimos que solucionar. Para ello hemos realizado el siguiente proceso:

En cuanto a la creación del corpus el proceso ha tenido las diferentes etapas agrupadas

en dos partes entrenamiento y evaluación:

- Entrenamiento:
  - Creación de la estructura arbórea del Cast3LB (árboles de constituyentes)
  - Búsqueda e implementación de la transformación de árboles de constituyentes en árboles de dependencias. Aquí realizamos un proceso de búsqueda de reglas de dependencias basándonos en el etiquetado de categorías gramaticales del Cast3LB. Este proceso ha sido de gran creatividad que nos ha llevado incluso a la realización de varios artículos científicos.
  - Estudio del Maltparser y búsqueda de un modelo adecuado para su entrenamiento.

  Evaluación:

  - Transformación manual de resultados en árboles.
  - Análisis de resultados y anotación de los casos particulares

La primera parte del proceso es la que nos va a ocupar en esta sección. En ella se detalla la creación de la estructura de los árboles de constituyentes, que serán posteriormente transformados por medio de un algoritmo de traducción de constituyentes a dependencias.

El corpus Cast3LB está compuesto por muchos archivos XML, y en cada archivo hay decenas de frases anotadas tanto semántica como sintácticamente. En esta primera etapa nos vamos a centrar en ir recorriendo los archivos XML y a medida que los vamos leyendo, guardamos la información de las palabras y sus etiquetas en la estructura de datos mencionada, que será posteriormente manipulada.

Esto lo hemos realizado con el API DOM (Document Object Model). El Document Object Model (en español Modelo de Objetos de Documento), frecuentemente abreviado DOM, es una forma de representar los elementos de un documento estructurado (tal como una página web HTML o un documento XML) como objetos que tienen sus propios métodos y propiedades. El responsable del DOM es el World Wide Web Consortium (W3C).

Mediante el API DOM conseguimos ir leyendo fácilmente los archivos XML del corpus y almacenando la información que vamos a necesitar. En la siguiente figura vamos a ver la estructura que va a tener cada uno de los nodos de cada árbol de constituyentes del corpus de entrenamiento.

4.3.1.2. Transformación algorítmica de árboles de constituyentes en árboles de dependencias

Hay principalmente dos formalismos para representar la estructura sintáctica de una frase: constituyentes (o de estructura de frase) y dependencias. Ambos tipos de gramáticas utilizan árboles para representar la estructura de una frase, ya que el significado de los nodos y de los arcos en cada árbol es distinto.

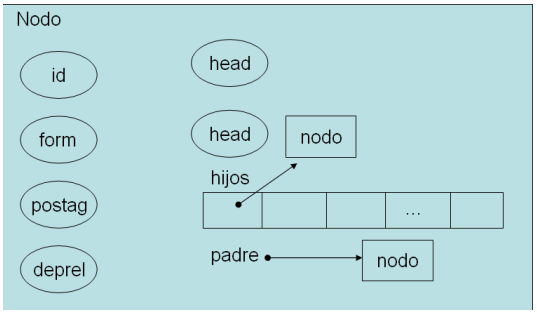

Figura 4.4: Estructura de un nodo

En la gramática de constituyentes (de estructura de frase), los nodos del arbol pueden ser la propia palabra, los diferentes sintagmas (nominales, preposicionales), grupo verbal, la propia palabra o la categoría léxica de la palabra, etc...Los arcos describen una relación de inclusión.

En los árboles de dependencias los nodos son las palabras, de modo que se establece una relación de dependencia entre cada par de palabras: una de las palabras es la principal o la que gobierna y la otra es la subordinada o dependiente de la primera.

Necesitamos una técnica heurística para, a partir de un corpus anotado sintácticamente dentro del formalismo de constituyentes obtener un corpus anotado sintácticamente dentro del formalismo de dependencias.

El método consiste en la extracción de una gramática libre de contexto (CFG) del corpus etiquetado y la identificación automática del elemento rector en cada regla, para poder construir el árbol de dependencias.

El proceso de transformación se realiza en varias etapas:

- Extracción de las reglas de constituyentes de la gramática a partir del Corpus Cast3LB.

Simplificación del corpus de constituyentes: las etiquetas del corpus Cast3LB se dividen en dos grupos. El primero especifica el "part-of-speech" (postag) como por ejemplo, nombre, verbo, sintagma nominal, grupo verbal, etc. Es la parte mas importante del etiquetado para nosotros. El segundo grupo de etiquetas especifica características adicionales, como pueden ser género y número. Estas características pueden omitirse para reducir el número de reglas de la gramática sin afectar a la transformación.

Para reducir el número de patrones de la gramática resultante, también se ha simplificado el etiquetado del Corpus Cast3LB eliminando los símbolos de puntuación.

Para extraer las reglas de la gramática, consideramos los nodos con mas de un hijo como la parte izquierda de una regla, y sus hijos serán la parte derecha.

- Marcado de la cabeza (head)

  - Después de haber realizado la extracción de todos los patrones de la gramática, marcamos automáticamente la cabeza (head) de cada patrón mediante la aplicación de métodos heurísticos. Marcamos la cabeza con el símbolo @.

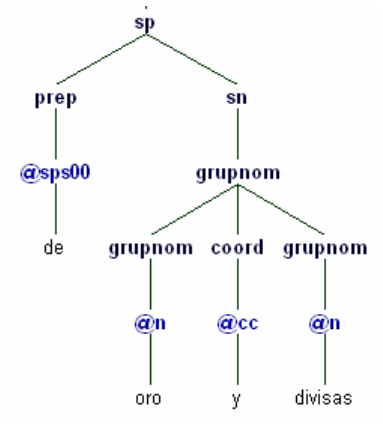

Figura 4.5: Nodos que tienen una hoja marcada como cabeza

```
(sp
  (prep
    (@sps00 de de))
  (sn
    (grupnom
      (grupnom
        (@n oro oro))
      (coord
        (@cc y y))
      (grupnom
        (@n divisas divisa)))))
```

Figura 4.6: Nodos que tienen una hoja marcada como cabeza

- La heurística que hemos empleado es la siguiente:
  - Si la regla contiene un solo elemento (o si solo uno de sus elementos puede ser cabeza) entonces es la cabeza:
    `grupnom =>@n`
  - Si el patrón tiene una coordinada (coord), entonces ésta es la cabeza:
    `grupnom =>grupnom @coord grupnom`
    `S =>@coord sn gv sn`
  - Si el patrón tiene dos o más coordinadas, la primera es la cabeza:
    `S =>@coord S coord S`
    `Sp =>@coord sp coord sp`
  - Si el patrón tiene un grupo verbal (gv) entonces éste es la cabeza:
    `S =>sn @gv sn`
    `S =>sadv sn @gv S Fp`
  - Si el patron tiene un pronombre relativo, entonces es la cabeza:
    `sp =>prep @relatiu`
    `sn =>@relatiu grupnom`
  - Si el patrón tiene una preposición (prep) como primer elemento y va seguida de sólo un elemento, entonces la preposición es la cabeza:

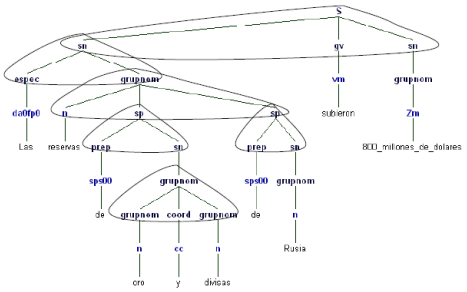

Figura 4.7: Patrones que serán extraídos de la oración "Las reservas de oro y divisas de Rusia subieron 800_millones_de_dolares"

```
sp =>@prep sn
sp =>@prep sp
```

- Si el patrón tiene un verbo infinitivo, entonces éste es la cabeza:

```
S =>@infinitiu S sn
S =>conj @infinitiu
S =>neg @infinitiu sa
```

- Si el patrón tiene un presente participio (gerundi), entonces éste es la cabeza:

```
S =>@gerundi S
```

- Si el patrón contiene un verbo principal (vm), éste es la cabeza:

```
gv =>va @vm
infinitiu =>va @vm
```

- Si el patrón tiene un verbo auxiliar (va) y otro verbo cualquiera, entonces el verbo auxiliar nunca es la cabeza:

```
gv =>va @vs
```

- Si el patrón tiene un especificador (espec), como primer elemento, núnca será la cabeza:

```
sn =>espec @grupnom
sn =>espec @sp
```

- En patrones cuyo padre sea un sintagma nominal, si el patrón contiene un nombre (n) entonces es la cabeza:

```
grupnom =>s @n sp
grupnom =>@n sn
grupnom =>s @n S
```

- En patrones cuyo padre sea un sintagma nominal (grupnom), si el patrón tiene un grupo nominal, entonces es la cabeza:

```
grupnom =>@grupnom s
grupnom =>@grupnom sn
```

- En patrones cuyo padre sea un especificador (espec), si el patrón contiene un artículo definido (da), entonces es la cabeza:

```
espec =>@da di
```

grupnom ← grupnom coord grupnom
sp ← prep sn
grupnom ← n sp sp
sn ← espec grupnom
S ← sn gv sn

Figura 4.8: Patrones extraídos de la oración "Las reservas de oro y divisas de Rusia subieron 800_millones_de_dolares"

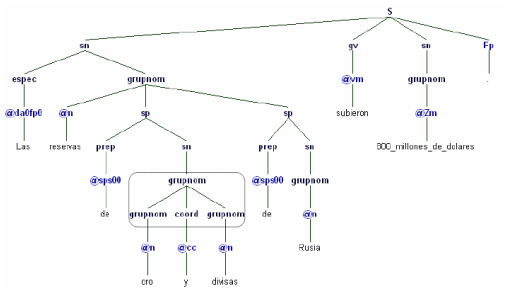

Figura 4.9: Árbol de constituyentes de la oración "Las reservas de oro y divisas de Rusia subieron 800_millones_de_dolares"

```
espec =>@da dn
```

- Si el patrón tiene un adjetivo calificativo (aq) y un sintagma preposicional (sp), entonces el adjetivo es la cabeza:

```
S =>sadv @aq sadv
sa =>sadv @aq sp sp
```

- Si las heurísticas que acabamos de enumerar no nos permiten determinar la cabeza sin ninguna ambigüedad, elegimos el primer elemento que puede ser cabeza.

- Utilizar recursivamente esta información de las cabezas para la transformación

  - El algoritmo de transformación usa recursivamente los patrones con las cabezas ya marcadas para determinar que componentes subirán en el arbol. Esto supone desconectar las cabezas de sus hermanos y ponerlas como nodo padre.

  - Describimos el algoritmo en detalle;
    - Recorremos el arbol de constituyentes en profundidad de izquierda a derecha, empezando por la raíz y visitando recursivamente los nodos hijos.
    - Para cada patrón del árbol buscamos las reglas para averiguar qué elemento es la cabeza.
    - Marcamos la cabeza en el arbol de constituyentes. Lo desconectamos de sus hermanos y lo colocamos en la posición del nodo padre
  - El algoritmo termina cuando un nodo cabeza acaba siendo la raíz.

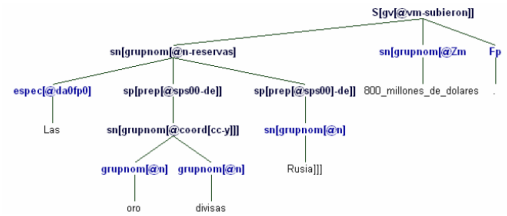

Figura 4.10: Árbol de dependencias resultante etiquetado

4.3.1.3. Obtencion de archivos de entrada para MaltParser a partir de la estructura interna de los árboles de dependencias

Para el tratamiento interno de los datos hemos creado una estructura basada en nodos. En la sección anterior se explicaba la transformación de árbol de constituyentes en árbol de dependencias. Ya desde el principio cada palabra de una oración está almacenada en un nodo que contendrá los siguientes atributos:

1. id = Identificador único dentro de la frase (entero).
2. form = Forma léxica de la palabra (String).
3. postag = Etiqueta del part-of-speech o categoría léxica.
4. head = Cabeza sintáctica (identificador de palabra). Palabra de la que depende.
5. deprel = Relación de dependencia con la cabeza.
6. puntero a padre
7. puntero a un ArrayList con sus nodos subordinados.

Cada oración estará representada mediante un conjunto de nodos como el descrito, siendo cada nodo una palabra. Este conjunto lo manejaremos mediante arrays de Java, los ArrayList. Un corpus estará compuesto por un ArrayList cuyos elementos son nodos con las palabras raíz de cada oración. De cada nodo "cuelga" un árbol con su oración. El atributo puntero a padre apuntará a null y el de los hijos contendrá un ArrayList con todos los hijos subordinados a ala raíz. Así recursivamente. En la figura 4.13 se muestra la estructura de un árbol. En la figura 4.11 se muestra la estrucutura de un nodo. En la figura 4.12 se ve como quedaría un árbol.

La idea es traducir todas las frases partiendo de nuestra estructura interna y depositarlas en un archivo con uno de los formatos que admite MaltParser, el MaltTAB. Además de la traducción a MaltParser, se utiliza este algoritmo para igualmente traducir la estructura y generar archivos DOT para la posterior generación de dibujos de los grafos correspondientes.

A continuación pasamos a explicar el algoritmo seguido para la traducción. El algoritmo que se usa es un bucle que recorre todos los nodos del array y para cada elemento (cada frase) se llama al método que traduce directamente una única frase. Este método lo que hace es transformar el árbol de la frase en un String con todo el contenido del

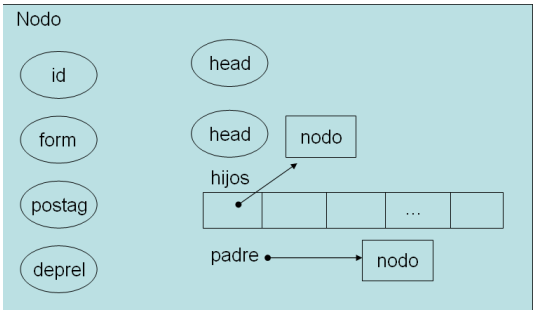

Figura 4.11: Estructura de un nodo

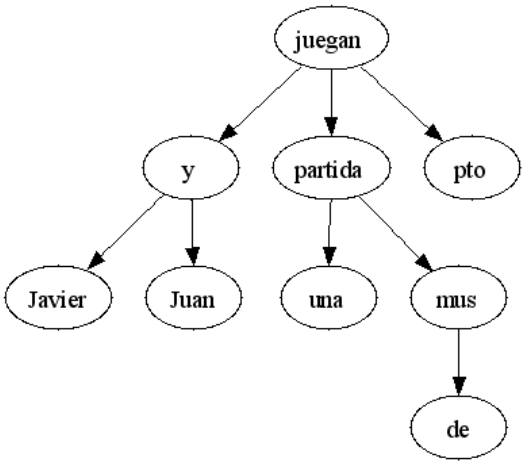

Figura 4.12: Ejemplo de nuestra estructura.

archivo .tab de esa frase en el correspondiente formato MaltTAB. En cada iteración del bucle (transformación de cada una de las frases) se va añadiendo un string al final de tal forma que cuando acaba el bucle queda un string con todo el contenido del archivo .tab. A continuación se crea un archivo con el nombre deseado por el usuario con la extensión .tab, donde se escribe el String resultante de la transformación de todas las frases. Éste archivo es el que se usará para invocar a MaltParser en modo entrenamiento. En la figura 4.14 se muestra el pseudocódigo del algoritmo.

4.3.2. Entrenamiento

Una vez que tenemos traducido los árboles de constituyentes en árboles de dependencias tenemos datos para poder entrenar a la herramienta de aprendizaje automático Maltparser.

Hemos elegido esta herramienta por la facilidad de su integración y su elevada precisión. Además, dicha herramienta nos permite entrenar varios modelos de forma independiente para luego realizar el análisis sobre el modelo que queramos.

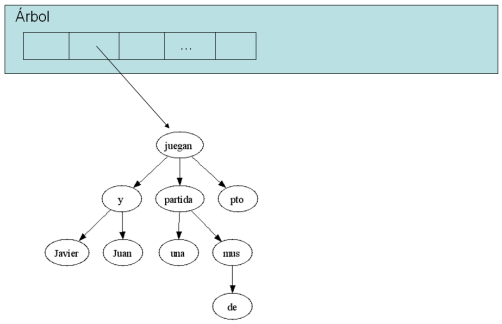

Figura 4.13: Estructura de un árbol

```
proc imprimir_hijos()
Mientras (arbol!=null) hacer
   Para cada hijo de arbol hacer
      Actual =  hijo;
      Añadir actual al archivo de texto;
      Para cada hijo de actual hacer imprimir_hijos();
   finPara
finMientras;
```

Figura 4.14: Pseudocódigo

4.3.2.1. Objetivos del entrenamiento

No hay que perder de vista que el objetivo del entrenamiento es la creación de un modelo de ánalisis de dependencias por parte del Maltparser, este modelo no basta con que pueda realizar árboles de dependencias a partir de un texto etiquetado, sino que los resultado del análisis han de ser suficientemente buenos como para que tenega sentido su uso práctico.

4.3.2.2. Uso del Maltparser en modo LEARNING

Como se explicó en el Capitulo 3, uno de los recursos utilizados es MaltParser. Para el entrenamiento es necesario pasar un archivo de opciones donde se especifican todos los flags para la ejecución de MaltParser como desee el usuario. Los flags necesarios para el aprendizaje son los especificados en la tabla 4.1, y que en su momento los introdujo el usuario en el menú Settings.

Cuando desde JBeaver invocamos MaltParser, se le pasa el archivo de opciones generado, mostrando en un panel de los Settings lo que responde por pantalla MaltParser. Los resultados obtenidos, se almacenarán donde lo especificó en su momento el usuario. El modelo generado por MaltParser se almacenará en la correspondiente carpeta de la aplicación. Este modelo será el que más tarde se use para poder realizar un análisis.

Parámetros	Flag	Descripción	Valores	Descripción
LEARNER	-l	Tipo de aprendizaje	MBL* SVM	Aprendizaje basado en memoria Máquina de soporte vectorial
LEARNEROPTIONS	-L	Parámetros de configuración	String	Opciones propias de TiMBL
		Opciones extra	-S [0123]	Estrategia para dividir los datos de entrenamiento para entrenar clasificadores SVM 1 = separación binaria cuando el token de la cima de la pila tiene HEAD = 0 2 = separación de acuerdo al valor de la característica del modelo (especificado por el flag -F) 3 = combinación de 1 y 2

Cuadro 4.1: Tabla con los flags necesarios para LEARNING

4.3.3. Análisis

Una vez tenemos entrenado un modelo, estamos en condiciones de realizar análisis de textos con dicho modelo. El ánáisis tiene dos partes: Por un lado la creación del propio árbol, es decir, marcar cuál es el padre de cada token, y por otro el marcado de la dependencia que entre padre e hijo hay.

4.3.3.1. Objetivos del análisis

Las dependencias nos dan mucha información sobre el contexto el significado de la frase etc... así que marcarlas correctamente es crucial. Además la reliazación del análisis puede tener objetivos, intenciones o usos muy diversos tales como: traducción automática, reconocimiento de voz, sistemas de preguntas-respuestas, representación intermedia para la recuperación de información...

4.3.3.2. Uso del Maltparser en modo PARSER

Cuando queremos analizar una oración con un cierto modelo previamente entrenado, lo que deberemos hacer es especificarlo así en el menú Settings. En este menú seleccionamos el archivo tab con la oración a analizar con formato MaltTAB. Otra opción, en lugar de pasar el archivo MaltTAB, es introducir una oración por el cuadro de texto de la pantalla principal. Esta opción se *parseará* y usando TreeTagger lo etiquetaremos con su postag, para escribirlo en un archivo y dárselo a MaltParser y que nos la analice. Los resultados del análisis se mostrarán en el panel de la izquierda, pudiendo ser seleccionadas para visualizar el resultado gráficamente.

Para el modo PARSER necesitaremos especificarle a MaltParser las opciones mediante un archivo. En la tabla 4.2 se muestran los flags necesarios para su correcta ejecución.

Parámetro	Flag	Descripción	Valores	Descripción
algoritmo	-a	tipo de algoritmo	NIVRE* COVINGTON	Nivre: Lineal O(n) Covington O(n^2)
opciones	-p	opciones de análisis	-a [ES]	Orden de arco (NIVRE): E(ager), S(tandard)
			-o [0123]	Oracle (NIVRE): 0 = por defecto (MaltParser 0.2) 1 = desplaza antes de reducir 2 = 1 + permite reducción de los TOKENS no-adjuntos (HEAD = 0) 3 = 2 + permite a las raices ser etiquetadas con DEPREL≠ROOTLABEL

Cuadro 4.2: Flags para la ejecución en modo PARSER

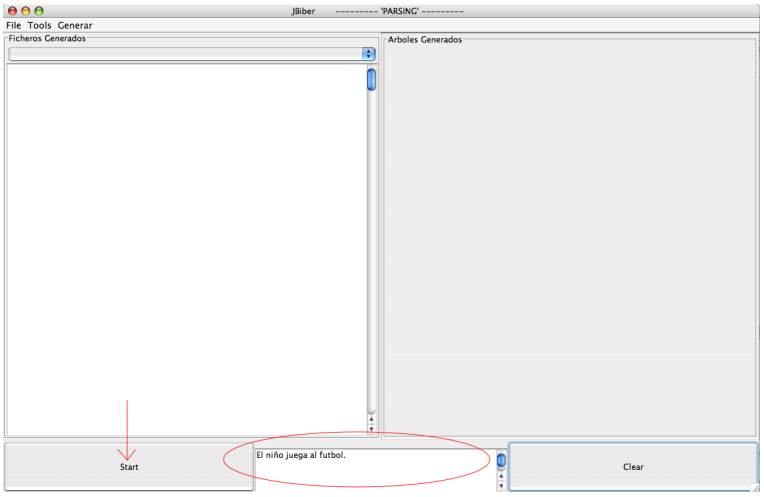

Figura 4.15: Introducción de una oración para su análisis

4.3.3.3. Proceso gráfico del análisis

Para mostrar un ejemplo de este proceso adjuntamos a continuación unas cpaturas de pantalla (Figuras 4.15 y 4.16) que muestran mejor la funcionalidad del análisis. En la Figura 4.15 , como vemos en el título de la ventana, el usuario ha cargado el modo "Parsing", esto se hace pro medio del menú de la barra Tools->View. Por esta razón le aparece un área de texto donde introducir el texto a analizar. Posteriormente sólo tiene que pulsar la tecla START. En la figura 4.16 el programa nos muestra un desplegable (1) con todas las oraciones del texto que en este caso solo hay una. En otra área de texto (2) podemos ver las oraciones ya analizadoas y en formato *.tab. Tras seleccionar en el desplegable la oración a visualizar solo tenenmos que ir al menú de la barra Generar para obtener el gráfico (3).

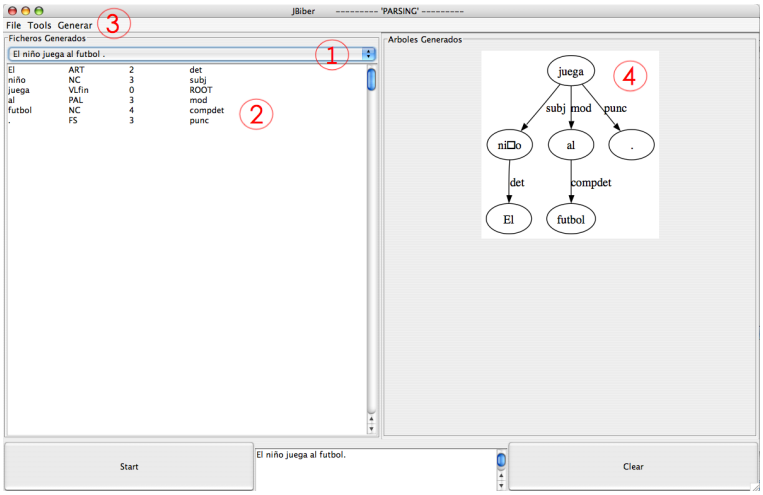

Figura 4.16: Ventana de visualización de los resultados

4.3.4. Evaluación

Como ya se comentó en la sección 3 del Capítulo 2 (Métricas de evaluación); CoNLL-X Shared Task proporciona una serie de programas en PERL para la evaluación de los resultados. Nuestro programa tiene dependencias propias y el etiquetado es dependiente de la aplicación treetagger por lo que no los hemos usado, pero sí la definición de su metodología que nos ha llevado a ampliar la funcionalidad de nuestro sistema incluyendo un programa que evalúe los resultados con las diversas métricas. Este programa no es trivial pues tanto la comprobación del correcto marcado de dependencias como de la correcta formación de los árboles, están basados en los algorítmos que se han utilizado para la creación de los corpus y que entre otras cosas nos ha llevado a la publicación de dos artículos científicos que versan en ello.

Las diferentes métricas de evaluación de resultados, que se explican en profundidad en el Capítulo 2, son: Labeled attachment score (LAS), Unlabelled attachment score (UAS), Unlabelled attachment score(UAS) y Label accuracy. En nuestro sistema hemos las hemos utilizado todas. Para realizar estos cálculos hay que hacer otros previos, así los resultados que mostramos son: Frases procesadas, Palabras procesadas, Palabras efectivas (las palabras que pertenecen a la categoría de símbolo Unicode, no computan para realizar las estimaciones), Número de dependencias, Número de dependencias desconocidas (el sistema no las encuentra, las marca por definición DEF), Número de dependencias bien calculadas, Número de Head (Número de cabezas a cada palabra que se tienen en cuenta para las estimaciones), Número de Head correctas, Número de Dep-Head correctas (Cada token tiene un Head y Dependencias, puntuamos cuando el par de una palabra está bien calculado). Hecho esto podemos calcular los porcentajes LAS, UAS, y Label accurancy como ya se han explicado en la sección 3 del Capítulo 2.

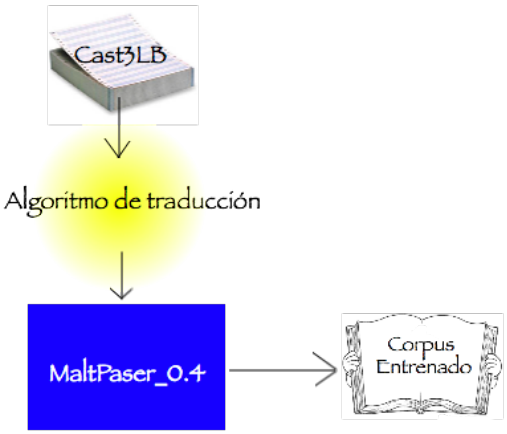

Figura 4.17: Proceso de creación del corpus de entrenamiento

4.3.4.1. Resultados

A continuación mostramos resultados de la evaluación de los diferentes corpus creados. Por un lado tenermos el Corpus de entrenamiento, que es con el que hemos entrenado la herramienta de aprendizaje Maltparser_0.4 y por otro los corpus de test ,que son corpus creados a partir de diferentes textos, a saber: Política, Economía, Medicina, Deporte y Justicia.

Al fijamos en estas tablas nos hacemos una idea de la calidad de nuestro sistema., debido a la gran precisión dada por las métricas LAS, UAS y Label accuracy

- Corpus de entrenamiento

El corpus de entrenamiento es el componente esencial de nuestro sistema pues los mayores esfuerzos del grupo de trabajo se han centrado en desarrollarlo de la mejor manera posible. En la Figura 4.17 vemos como el corpus de entrenamiento es el resultado de todo un proceso. Al ser el corpus con el que entrenamos el Maltparser_0.4 tiene que tener más calidad pues de él depende que las evaluaciones sobre los corpus posteriores se realicen correctamente. Esto es fácil de ver observando un momento, en la Figura 4.18, los datos con porcentajes muy altos.

En la Figura 4.18 se observan los datos de la evaluación del corpus de entrenamiento. Destacar las tres medidas utilizadas labeled attachment score (LAS), Unlabelled attachment score (UAS) y Label accuracy.

RESULTADOS	TIPO DE TEXTO
Frases procesadas: 779 Palabras procesadas: 20859 Palabras efectivas: 18542 Numero de Dep : 18542 Numero de Dep desconocidas: 1895 Numero de Dep bien calculadas: 16554 Numero de Head : 18542 Numero de Head correctas: 14967 Numero de Dep-Head correctas: 13780 Labeled attachment score (LAS): 74.32 % Unlabelled attachment score (UAS): 80.72 % Label accuracy: 89.28 %	Corpus de entrenamiento

Figura 4.18: Resultados del corpus de entrenamiento

- Corpus de test

En las páginas siguientes se muestran las tablas que contienen los resultados del análisis de diferentes tipos de textos, a saber: política, medicina, economía, justicia y deporte.

RESULTADOS	TIPO DE TEXTO
Frases procesadas: 13 Palabras procesadas: 548 Palabras efectivas: 503 Numero de Dep : 503 Numero de Dep desconocidas: 30 Numero de Dep bien calculadas: 414 Numero de Head : 503 Numero de Head correctas: 370 Numero de Dep-Head correctas: 325 Labeled attachment score (LAS): 64.61 % Unlabelled attachment score (UAS): 73.56 % Label accuracy: 82.31 %	Política
Frases procesadas: 10 Palabras procesadas: 313 Palabras efectivas: 282 Numero de Dep : 282 Numero de Dep desconocidas: 15 Numero de Dep bien calculadas: 235 Numero de Head : 282 Numero de Head correctas: 216 Numero de Dep-Head correctas: 190 Labeled attachment score (LAS): 67.38 % Unlabelled attachment score (UAS): 76.6 % Label accuracy: 83.33 %	Medicina
Frases procesadas: 27 Palabras procesadas: 997 Palabras efectivas: 899 Numero de Dep : 899 Numero de Dep desconocidas: 60 Numero de Dep bien calculadas: 769 Numero de Head : 899 Numero de Head correctas: 685 Numero de Dep-Head correctas: 599 Labeled attachment score (LAS): 66.63 % Unlabelled attachment score (UAS): 76.2 % Label accuracy: 85.54 %	Economía

RESULTADOS	TIPO DE TEXTO
Frases procesadas: 21 Palabras procesadas: 1726 Palabras efectivas: 1588 Numero de Dep : 1588 Numero de Dep desconocidas: 120 Numero de Dep bien calculadas: 1220 Numero de Head : 1588 Numero de Head correctas: 1162 Numero de Dep-Head correctas: 957 Labeled attachment score (LAS): 60.26 % Unlabelled attachment score (UAS): 73.17 % Label accuracy: 76.83 %	Justicia
Frases procesadas: 18 Palabras procesadas: 596 Palabras efectivas: 550 Numero de Dep : 550 Numero de Dep desconocidas: 42 Numero de Dep bien calculadas: 431 Numero de Head : 550 Numero de Head correctas: 422 Numero de Dep-Head correctas: 338 Labeled attachment score (LAS): 61.45 % Unlabelled attachment score (UAS): 76.73 % Label accuracy: 78.36 %	Deporte

4.3.5. Visualización de resultados

Una de las características de JBeaver es que una vez transformado un corpus de entrenamiento, o cuando analizamos alguna oración bien por teclado bien por archivo, se muestran en el panel de la izquierda todas y cada una de las oraciones analizadas. Para comprobar la corrección del análisis decidimos que sería bueno representar las oraciones mediante un árbol de dependencias. En el panel de la derecha se van generando y acumulando los árboles generados. También existe la posibilidad, mediante un comando del menú, de mostrar la creación dinámica del árbol de una oración. Este comando crea una nueva ventana de la aplicación que muestra la creación de las ramas con un retardo suficiente para seguirlo. También posee un botón para parar o reanudar la generación de ramas, otro para dar un paso manualmente, y otro para terminar la generación del árbol de golpe y añadir el árbol generado en el panel que acumula los grafos de las oraciones. En la figura 4.19 se muestran una sucesión de los contenidos de los archivos generados en los sucesivos pasos. Y en la figura 4.20 se ven los archivos resultantes.

La generación de los archivos DOT para la visualización de las estructuras de las oraciones se hace mediante el mismo algoritmo que usamos para traducir la estructura interna de oraciones a formato MaltTAB. En la seccion 4.3.1.3 se explica este algoritmo.

```
digraph 0 {
es[label="es"];
}
```

```
digraph 0 {
es ->perro[label="SUJ"];
}
```

```
digraph 0 {
es ->perro[label="SUJ"];
es ->marron[label="ADV"];
}
```

```
digraph 0 {
es ->perro[label="SUJ"];
es ->marron[label="ADV"];
es ->pto[label="PTO"];
}
```

```
digraph 0 {
es ->perro[label="SUJ"];
es ->marron[label="ADV"];
es ->pto[label="PTO"];
perro ->El[label="DET"];
}
```

Figura 4.19: Sucesión de archivos.

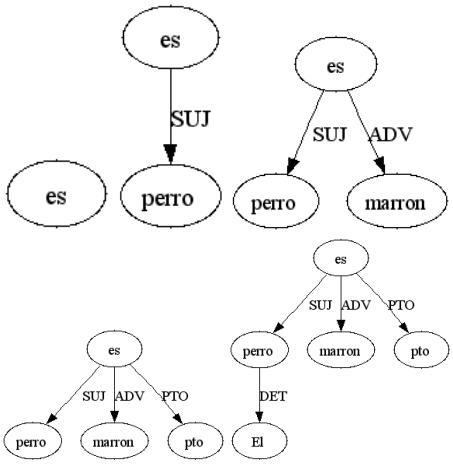

Figura 4.20: Archivos resultantes de la figura 4.19

La única diferencia es que en el paso de escribir en el String, lo hacemos con formato DOT, y en lugar de almacenar todo en un único archivo, lo almacenamos en archivos independientes. El objetivo de almacenarlos en archivos distintos es porque a la hora de convertirlo en imagen podamos cogerlos por separado y no siempre generar todos los grafos en imagenes a la vez. Otro de los motivos es que se saturaría la aplicación dot para intentar generar, en el peor de los casos una cantidad considerable de gráficos. Así en cada iteración del bucle externo, creamos un archivo que almacenamos en el directorio ./resultados/grafos de la ruta de la aplicación.

Para la opción de la ventana emergente, lo que se hace es generar nuevos archivos DOT a partir de los generados para cada oración. Al invocar la opción de visualización "por partes", lo que hacemos es mediante el identificador de la oración seleccionamos el archivo DOT correspondiente. Este archivo DOT, mediante un bucle, lo *parsearemos* para ir generando poco a poco archivos temporales del grafo. Cada nuevo archivo representa el grafo con un paso más, es decir con el siguiente brazo a dibujar. Estos grafos que vamos generando poco a poco son los que se irán dibujando en cada paso.

Capítulo 5

Conclusiones y trabajo futuro

A la hora de realizar la especificacion de JBeaver nos planteamos una serie de objetivos de diversa índole. JBeaver surge como de un proyecto de investigacion en el que desarrollar un analizador de dependencias para el español utilizando la herramienta de aprendizaje automático *Maltparser*.

MaltParser cubre todas las necesidades que tenemos, ya que es autónomo, fácil de integrar (se invoca desde línea de comandos, como *Minipar*) y proporciona unos resultados mas que notables para las diferentes lenguas en las que se ha empleado para el análisis de dependencias.

Hemos conseguido que el sistema sirva para entrenar de manera sencilla modelos de Maltparser, pudiendo hacerse extensible como analizador de dependencias para cualquier idioma.

Se han conseguido los objetivos propuestos en la especificación, se ha obtenido una herramienta fácil de instalar y utilizar con un rendimiento acorde con los sistemas actualmente existentes.

El objetivo final era un analizador de dependencias para el español, de libre distribución y que fuera fácil de instalar y manejar.

5.1. Conclusiones

JBeaver, el resultado del proyecto, es un sistema autonomo, disponible gratuitamente para su ejecucion en los principales sistemas operativos, facil de instalar y utilizar, y con alto rendimiento para el analisis de dependencias del español. Al estar implementado en Java, el unico recurso que ha de estar previamente instalado en el computador es el Java Runtime Envorinment 5.0. Aunque el enfoque de la funcionalidad como analizador de dependencias de JBeaver es muy similar al del sistema desarrollado por el grupo de Nivre [23], ya que JBeaver hace uso de Maltparser, tres son las principales aportaciones que diferencian a JBeaver:

1. No es un prototipo de laboratorio, sino que esta disponible publicamente en un paquete jar, muy simple de instalar y utilizar.

2. No solo realiza analisis de dependencias de textos, mediante el modelo previamente entrenado de Maltparser incluido en la distribucion, sino que cada usuario puede entrenar un modelo de Maltparser con el que ejecutar posteriormente analisis desde el propio JBeaver.

3. Posee una interfaz graca de usuario para el entrenamiento de Maltparser y para el analisis de textos, con salida visual de los arboles de analisis generados. Pero, ademas, puede ser invocado en modo no graco, desde la consola de comandos del sistema operativo, realizando la entrada/salida mediante cheros de texto, y analizando mediante el modelo de Maltparser que incluye la distribucion.

Así mismo, por los resultados obtenidos en las pruebas de evaluacion a que ha sido sometido, se puede considerar a JBeaver como una interesante opción para el analisis de dependencias para el español. Ademas, JBeaver es virtualmente un analizador de dependencias para cualquier idioma, ya que se puede utilizar para entrenar modelos de Maltparser que posteriormente pueden ser utilizados para el analisis. Pero hay que tener en cuenta que aunque Maltparser ha sido ya probado para varios idiomas, no se puede asegurar todava su universalidad como analizador de dependencias..

5.2. Trabajo futuro

Muchos son los trabajos futuros que se pueden realizar sobre nuestro sistema, pues su caracter modular le otorga una gran flexibilidad. Ya existen diversos estudios que esperamos se implementen a corto-medio plazo como son: Hacer del sistema un analizador prácticamente miltilingüe creando modelos de análisis para un gran número de idiomas. Además hay conjuntos de oraciones particulares como pueden ser las subordinadas que requieren de un tratamiento especial y que ya tenermos un estudio teórico de las mismas pero esperamos incluir su implementación en un futuro inmediato. También hay propuestas para la Detección y corrección de análisis incorrectos, Introducción de entidades nombradas (Detecte conjuntos de palabras como una sola entidad Ej: Pepe Ruíz), y por último el ajuste del modelo de entrenamiento para poder mejorar el resultado de nuestro análisis creando un corpus de dependencias más eficiente.

5.2.1. Hacia un analizador de dependencias multilingüe.

En esta sección se proponen ideas básicas para realizar entrenamientos de JBeaver orientados a obtener analizadores de dependencias multilingües. Se espera que en un futuro próximo JBeaver aumente su funcionalidad con la inclusión de difentes idiomas, que entre otras cosas permita realizar estudios comparativos del comportamiento de las estructuras de dependencias en diversas lenguas.

5.2.1.1. Análisis de dependencias en otras lenguas con JBeaver

Se ha probado Maltparser para la generación de analizadores de dependencias en diferentes lenguas, con resultados notables, en la CoNLL Shared Task 2007. Igualmente,

JBeaver ha sido sometido a entrenamientos experimentales para obtener analizadores de dependencias para el griego moderno. Para ello se han utilizado corpora anotados de muy pequeño tamaño, en comparación con los habitualmente utilizados en tareas como la CoNLL Shared Task, del orden de 10 a 15 frases. A pesar del tamaño de los corpora, se han llegado a obtener resultados muy interesantes, como el que se muestra en la Figura 5.1:

Η Θεοδώρα	βλέπει	τηλεόραση	κάθε	πρωί.
Teodora	ve	la televisión	cada	mañana.

Figura 5.1: Oración en Griego y Castellano

Tras el entrenamiento, se analizó la frase anterior, obteniéndose el siguiente árbol de análisis, Figura 5.2, correctamente realizado:

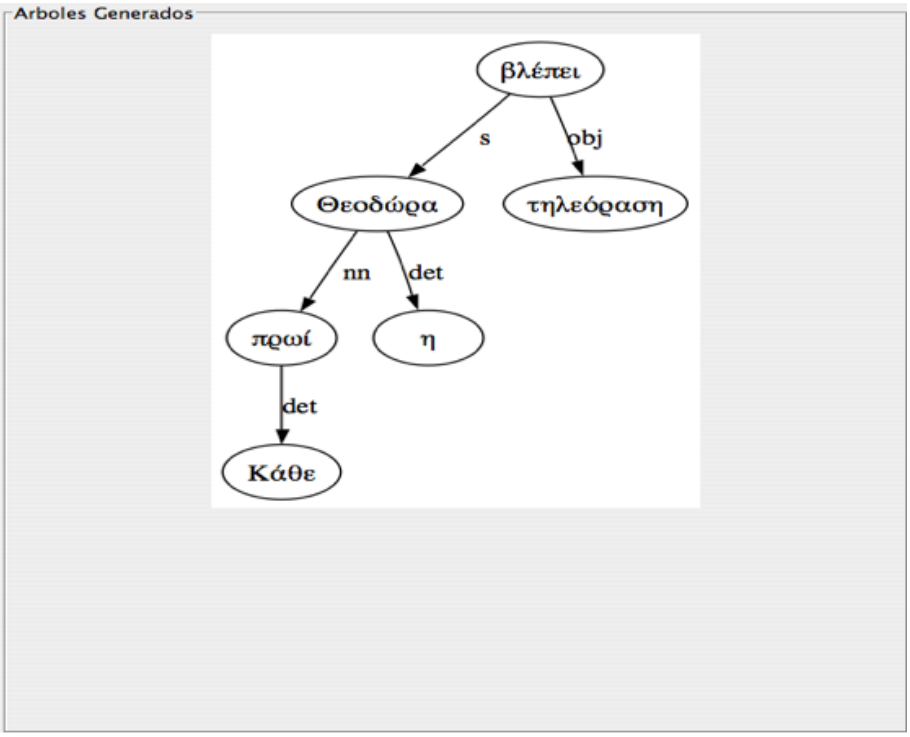

Figura 5.2: Árbol resultante de dependenicas

Actualmente se encuentra en estado inicial de desarrollo el entrenamiento de JBeaver para el análisis de dependencias del griego moderno, en colaboración con el Instituto para el Procesamiento del Lenguaje, (Ινστιτούτο Επεξεργασίας του Λόγου) http://www.ilsp.gr de Atenas.

5.2.1.2. Análisis de dependencias multilingüe con JBeaver

Hay lenguas, como el griego y el español, que presentan un alto paralelismo en sus gramáticas y, por tanto, unas estructuras sintácticas muy similares en un alto porcentaje de casos, como ocurre en el ejemplo visto en la subsección anterior. Esto puede significar que, escogiendo adecuadamente los ejemplos con los que realizar el entrenamiento de JBeaver, se podría obtener un analizador multilingüe. Esta idea se basa en que en el entrenamiento de JBeaver no importan las palabras utilizadas en los ejemplos contenidos en el corpus de entrenamiento, sino sólo las relaciones sintácticas entre ellas, así como sus categorías gramaticales. Por ello, eligiendo para cada ejemplo la estructura sintáctica más general que se pueda dar entre todas las lenguas para las que se quiera entrenar en simultaneidad JBeaver, se puede obtener un modelo con el que posteriormente analizar textos en todas las lenguas para las que se ha entrenado. Para ello se propone un análisis abstracto, en el que cada ejemplo del corpus de entrenamiento contenga las etiquetas que defininen las relaciones sintácticas entre las palabras, así como sus categorías gramaticales, cambiando las palabras concretas por símbolos abstractos que las representen. Por ejemplo, la frase *"Ο κύριος Χαρτ είναι από το Λονδίνο"* (El señor Hart es de Londres) y la frase "El doctor Marañón viene de la playa" presentan el mismo análisis de dependencias y, por tanto, podrían ser analizadas ambas entrenando un modelo con el ejemplo abstracto que se observa en la Figura 5.3

Nodo	Palabra abstracta	Categoría gramatical	Nodo rector	Función sintáctica
1	α	DET	3	det
2	β	NC	3	lex-mod
3	γ	NP	4	subj
4	δ	V	0	ROOT
5	ε	PREP	4	pred
6	ζ	DET	7	det
7	η	NP	5	pcomp-n
8	.	PF	0	punc

Figura 5.3: Componentes del análisis de dependencias

Es de esperar que con una cuidada selección de los ejemplos de entrenamiento se puedan obtener modelos para realizar análisis de dependencias multilingües.

5.2.2. Entrenamiento de oraciones complejas.

El sistema de análisis de dependencias JBeaver tiene un comportamiento excelente para oraciones no excesivamente complejas. Para oraciones compuestas el algoritmo de transforamción de árboles de constituyentes en árboles de dependencias no es trivial. No obstante conocemos el proceso y hemos dejado el código preparado para implementarlo.

Como ya se ha comentado en secciones anteriores, el algorítmo se basa en la extracción

de una gramática libre de contexto del corpus etiquetado, identificación automática del elemento rector en cada regla, y uso de esta información para la construcción del árbol de dependencias. Dichas reglas son "planas" (estructura de exactamente dos niveles). Por ello el algoritmo de transformación no da un tratamiento adecuado para el caso de los pronombres relativos, pero es importante hacerlo para mejorar los resultados del corpus de dependencias extraído a partir del corpus de constituyentes Cast3LB. Los pronombres relativos son palabras (que, cuyo, quien, etc.) que se refieren a otra anterior, antecedente, e introducen una oración finita relativa de la que forman parte. Para estos casos el algoritmo original encuentra el patrón `S.F.R <- relatiu gv sn`, marcando como rector el pronombre relativo quedando de la siguiente forma `S.F.R <- @relatiu gv sn`, y lo subiría en el árbol. Sin embargo, esta no es la estructura correcta.

Al conjunto de las heurísticas hay que agregar una más que marca pronombres relativos con esta marca: `@@relatiu`. El proceso de la aplicación de esta heurística especial se queda como sigue:

- Heurísticas con esta marca tienen prioridad sobre otras. Se aplica para marcar el pronombre: `S.F.R <- @@relatiu gv sn`.
- Se genera una regla con el elemento marcado removido: `S.F.R <- gv sn`.
- Se aplican las demás heurísticas en orden normal, para determinar el rector de tal patrón: `S.F.R <- @gv sn`.
- Se regresa el elemento con la marca especial: `S.F.R <- @@relatiu @gv sn`.

La regla resultante se interpreta para generar el subárbol: cuando la regla contenga el rector `@@`, el algoritmo lo coloca en la posición de su nodo padre; el rector `@` como un nodo hijo del rector `@@`, y los nodos no marcados los subordina al rector `@`.

5.2.3. Detección y corrección de análisis incorrectos

Si bien en este proyecto hemos realizado un estudio pormenorizado de los resultados obtenidos y hemos tratado de aumentar la eficiencia y precisión de los análisis; no se nos escapa que es un campo de estudio en el que no existe demasiada información y en el que si bien algunos lingüístas han mostrado gran interés, no está ni mucho menos extendido. De ahí que la comprobación de posibles casos erroneos en la formación de los árboles de dependencias, sea una tarea para expertos del idioma. Aunque hemos entablado contacto con algún que otro especialista, queda para posteriores esfuerzos la colaboración con los mismos para obtener unos resultados más documentados, fiables y que nos permitan avanzar.

5.2.4. Introducción de entidades nombradas

En ocasiones una estructura o conjunto de palabras representan a un solo ente. Por ejemplo: El conjunto de palabras "Pepe Pérez Gómez" identifica a una sola entidad y por tanto no tiene ningún sentido que creemos dependencias entre sus palabras para

con el resto de una posible oración. Por tanto si en nuestro análisis introducimos de la siguiente manera "Pepe_Pérez_Gómez" nuestro programa no solo identificará que es una única palabra, sino que además lo marcará como nombre propio. Un trabajo futuro sería que el propio programa identificara estas estructuras y las marcara como una única entidad lo que nos evitaría tener que introducir las oraciones de una cierta forma, pues son numerosas las estructuras que se repiten como fechas, nombres, siglas...

5.2.5. Ajuste del modelo de entrenamiento

La herramienta de aprendizaje que utilizamos, *Maltparser_0.4,* otorga a nuestra aplicación una gran potencia y versatilidad, esto es debido a que contiene numerosas opciones configurables a través de unos *flags* o parámetros. En el futuro un trabajo a realizar sería el ensayo y estudio del comportamiento del análisis de dependencias de nuestro programa variando dichos parámetros y fijar aquel con el que se obtengan mejores resultados.

Por otro lado *Maltparser_0.4* utiliza un modelo de pilas en el que podemos configurar el comportamiento del algorítmo de aprendizaje. En la actualidad hemos probado con varios que *Maltparser_0.4* pone a disposición del usuario; pero para trabajos posteriores se podrían crear modelos nuevos que se ajustaran a la lengua que queramos utilizar.

5.2.6. Mejora del tratamiento de palabras

A la hora de marcar las categorías gramáticales de cada palabra, nuestro sistema JBeaver se basa en el etiquetador *treetagger*. Este programa presenta problemas con palabras que terminan en vocal con tilde; pues las entiende como dos palabras diferentes. Por ello un trabajo futuro sería el particular tratamiento de este tipo de palabras para su correcta identificación.

Capítulo 6

Publicaciones y difusión del trabajo

Debido a la importante vertiente practica del proyecto, hemos creido que es muy importante cuidar los aspectos relacionados con la difusion, explicacion y, en definitiva, todo lo que ayude e informe a los potenciales usuarios de la aplicacion, que se dividen claramente en dos grupos:

- *Personal docente e investigador en el campo del Procesamiento del Lenguaje Natural*: Son firmes representantes de un sector en auge tanto en numero como en recursos destinados, sobre todo en la universidad española.
- *Profesores y alumnos de educacion primaria y secundaria:* Creemos firmemente que nuestra herramienta puede ayudar a una mejor comprension de una materia tan árida como puede ser la sintaxis de la lengua española. Puede ayudar a los profesores a explicar la materia y a los alumnos a comprender el análisis sintáctico desde el punto de vista del análisis de dependencias.

De cara al personal docente e investigador, hemos querido poner nuestro particular granito de arena en forma de dos artículos [1]:

- J. Herrera, P. Gervás, P.J. Moriano, A. Muñoz, L. Romero. 2007. *Building Corpora for the Development of a Dependency Parser for Spanish Using Maltparser.* Aceptado para el congreso de la SEPLN de 2007 [12].
- J. Herrera, P. Gervás, P.J. Moriano, A. Muñoz, L. Romero. 2007. *JBeaver: Un Analizador de Dependencias para el Español Basado en Aprendizaje.* Under evaluation process for CAEPIA 2007 [11].

[1] Los artículos pueden verse íntegros en el apéndice

Figura 6.1: Página Web de nuestro grupo de trabajo

Hemos creido conveniente integrar nuestra aplicacion en una página web Figura 6.1, actualmente en proceso de desarrollo, donde se incluya un ejecutable de nuestra aplicacion. La descarga de la aplicación se puede realizar de forma totalmente gratuita, ya que queremos contribuir a su mayor difusión.

Ahora que hemos explicado lo que nos ha motivado a escribir los artículos, vamos a entrar mas en profundidad sobre el contenido de dichos articulos.

Dos artículos se han enviado a la Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN). La SEPLN pretende difundir la investigación y el desarrollo realizado por los investigadores en el campo del Procesamiento del Lenguaje Natural (PLN), facilitando a la comunidad científica y empresarial del sector un foro idóneo para mostrar las posibilidades reales de aplicación del PLN.

La SEPLN además pretende el establecimiento de los canales adecuados para el intercambio de información y materiales científicos entre miembros, la organización de seminarios, simposios y conferencias, la promoción de publicaciones y la colaboración con otras instituciones de caracter nacional o internacional relacionadas con su ámbito de actuación.

La SEPLN organiza cada año un Congreso de dos o tres dias de duración. En él, tienen lugar sesiones científicas en forma de conferencias invitadas o sesiones técnicas, comunicaciones, demostraciones y mesas redondas sobre los diversos ámbitos propios del procesamiento del lenguaje natural (Análisis morfológico, sintáctico, semántico y pragmático, lexicografía computacional, traducción automatica, tecnología del habla, entornos de comunicación persona-máquina en lenguaje natural, gestión documental y lingüística de corpus).

Por ello, hemos creido que la SEPLN es uno de los foros adecuados en los que publicar nuestra pequeña gota de agua al inmenso mar que comprende el Procesamiento del Lenguaje Natural.

Capítulo 7

Agradecimientos

Esta memoria estaría incompleta si no hicieramos referencia a todos aquellos a los que estamos agradecidos por su participación directa o indirecta en el proyecto.

Para empezar, agradecer a Pablo todo el esfuerzo que ha dedicado para proporcionarnos tanto material y, sobre todo, tanta experiencia, que en estas situaciones es lo que uno agradece, ya que no sabe cómo lo está haciendo de bien.

También hacer mención a la gran ayuda que nos ha dado Jesús H. y a todo su equipo del departamento de Lenguajes y Sistemas Informáticos de la UNED, por facilitarnos el Corpus Cast3LB. Además agradecer el tiempo tanto físico como de cabeza dedicado a concretar reuniones, objetivos, dar ideas, buscar alternativas viables, ...

Evidentemente deberemos citar a Joakim Nivre y su compañeros de investigación, de la Universidad de Växjö en Suecia, por la creación de tan magnífica aplicación como es MaltParser. Animamos a toda persona interesada en el estudio del análisis sintáctico de dependencias que siempre los tenga como referencia.

Por otro lado agradecer al Grupo de investigación en Procesamiento del Lenguaje y Sistemas de Información, del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante, por el trabajo puesto en la creación del corpus Cast3LB.

Por último, agradecer a todos los compañeros de promoción que nos han soportado tanto en momentos de euforia como de depresión profunda, y nos han cubierto las espaldas para la entrega de prácticas y prestar apuntes... Gracias a Ignacio, Tony, Bruno, Ana, Miguel, Pablo, Guantxe, Felipe, Fernando, Pedro, Kike,...

Capítulo 8

Apéndice

8.1. Building corpora for the development of a dependency parser for spanish using maltparser

8.2. JBeaver Un Analizador de Dependencias para el Español

Bibliografía

[1] Daniel Hodges Ian Niles Adrian Novischi Jens Stephan Abraham Fowler, Bob Hauser. Applying COGEX to Recognize Textual Entailment.

[2] Elena Akhmatova. Textual Entailment Resolution via Atomic Propositions.

[3] Giuseppe Attardi. Experiments with a Multilanguage Non-Projective Dependency Parser.

[4] H. Calvo and A. Gelbuckh. In *DILUCT: An Open-Source Sapnish Dependency Parser based on Rules, Heuristics and Selectional Preferences*, pages 164–175, Klagenfurt, Austria, May 2006.

[5] A. Canisius, T. Bogers, A. van den Bosch, J. Geertzen, and E.T. Kin Sang. In *Dependency Parsing by Inference over High-recall Dependency Predictions*, New York, USA, June 2006.

[6] X. Carreras, M. Surdeanu, and L. Márquez. In *Projective Dependency Parsing with Perceptron*, New York, USA, April 2006.

[7] S. Corston-Oliver and A. Aue. In *Dependency Parsing with Reference to Slovene, Sapnish and Swedish*, New York, USA, June 2006.

[8] Lucy Vanderwendeand Deborah Coughlin and Bill Dolan. In *What Syntax can Contribute in Entailment Task*, 2005.

[9] G. Eryigit and K. Oflazer. In *Statistical Dependency Parsing of Turkish*, pages 89–96, Trento, Italy, 2006.

[10] Jesús Herrera, Anselmo Peñas, and Felisa Verdejo. In *Textual Entailment Recognision Based on Dependency Analysis and WordNet*, 2005.

[11] Pedro J. Moriano Alfonso Muñoz Luis Romero Jesús Herrera, Pablo Gervás. Building Corpora for the Development of a Dependency Parser for Spanish Using Maltparser.

[12] Pedro J. Moriano Alfonso Muñoz Luis Romero Jesús Herrera, Pablo Gervás. JBeaver: Un Analizador de Dependencias para el Español.

[13] Valentin Jijkoun and Maarten de Rijke. Recognizing Textual Entailment Using Lexical Similarity.

[14] M. jinshan, Z. Yu, L. Ting, and L. Sheng. In *Statistical Dependency Parsing of Chinese under Small Training Data*, Sanya City, Hainan Island, China, 2004.

[15] Jens Nilsson Gülsen Eryigit Svetoslav Marinov Joakim Nivre, Johan Hall. Labeled Pseudo-Projective Dependency Parsing with Support Vector Machines.

[16] Milen Kouylekov and Bernardo Magnini. Recognizing Textual Entailment with Tree Edit Distance Algorithms.

[17] Dekang Lin. In *Dependency-Based Evaluation of MINIPAR*, Granada, Spain, May 1998.

[18] Fabio Massimo Zanzotto Maria Teresa Pazienza, Marco Pennacchiotti. Textual Entailment as Syntactic Graph Distance: a rule based and a SVM based approach.

[19] Igor Mel'cuk. *Levels of Dependency in Linguistic Description: Concepts and Problems*, volume 1 of *Dependency and Valency. An International Handbook of Contemporary Research*. Berlin, Germany and New York, USA, 2006.

[20] Dan Roth Ming-Wei Chang, Quang Do. A Pipeline Model for Bottom-Up Dependency Parsing. 2006.

[21] Borja Navarro, P. Moraleda, B. Fernández, R. Marcos, and M. Palomar. In *Anotación de roles semánticos en el corpus 3LB*, Octubre 2004.

[22] Joakim Nivre, Johan Hall, and Jens Nilsson. In *MaltParser: A Data-Driven Parser-Generator for Dependency Parsing*, Genova, Italy, 2006.

[23] Joakim Nivre, Johan Hall, and Jens Nilsson. In *MaltParser: A Data-Driven Parser-Generator for Dependency Parsing*, March 2006.

[24] Joakim Nivre, Johan Hall, Jens Nilsson, G. Eryigit, and S. Marinov. In *Labeled Pseudo-Projective Dependency Parsing with Support Vector Machines*, New York, USA, June 2006.

[25] Manuel Pérez-Coutiño, Manuel Montes y Gómez, Aurelio López-López, Luis Villaseñor-Pineda, and Aarón Pancardo-Rodríguez. In *A Shallow Approach for Answer Selection based on Dependency Trees and Term Density*, 2006.

[26] G. R. Sampson. In *English for the Computer*, 1995.

[27] Lucien Tesnière. *Éléments de Syntaxe Structurale*. Paris, 1976.

[28] Yu-Chieh Wu Yue-Shi Lee Jie-Chi Yang. The Exploration of Deterministic and Efficient Dependency Parsing.